The Pragmatic Programmers

TURING 图灵程序设计丛书

Dr.Dobb's Jolt 第18届Jolt生产效率大奖图书

发布！软件的设计与部署

通过实例探讨如何构建稳健、务实的软件架构



【美】Michael T. Nygard 著　涂鸣 译

人民邮电出版社
POSTS & TELECOM PRESS

“敏捷开发强调在每次迭代结束时交付产品级代码，这本书准确展示了这一要求对当今重要系统的真正含义，你一定会获益匪浅。”

——Tom Poppendieck，Poppendieck.LLC

“卓尔不群，与众不同！这本书绝对会为大公司在一次项目发布中节省几十万甚至数百万的经费。”

——Jared Richardson，Agile Artisans公司

“非常值得关注！作者将20余年的经验、见解和总结的模式倾囊相授，指出了你已经犯的却还未意识到的所有错误。更令人振奋的是，他还给出了补救的诀窍。这本书对架构师以及所有程序员而言必不可少。”

——Arun Batchu，百思买全球电子商务产品开发部主管

“不管你是否为大型电商组织工作，书中的内容都会让你受益匪浅。”

——Christophe Addinquy，Zenika公司敏捷教练

“这本书对我发布软件影响巨大！写得很好，案例详实，开发运营必读。”

——Amazon.com读者评论

Release It!

Design and Deploy Production-Ready Software

“功能完成”并不意味着“产品就绪”，即便所有模块都通过了测试，也不足以证明软件就可以进行实际应用了。狂热的用户、超大的流量，以及闻所未闻的病毒，都可能是软件发布1.0版要面对的残酷现实。

本书针对以上挑战，详细展示软件发布前可能出现的种种问题以及相应的解决之道，适合包括网站和网络服务在内的所有企业级软件的架构师、设计师和开发人员阅读。书中所有主题都是通过作者自己研究过的真实案例来阐述的，具体内容分为四大部分：如何保证系统的稳定性，系统容量的真正含义以及优化方法，保证软件在数据中心流畅运行需要注意的设计问题，系统的运行寿命。

The Pragmatic Programmers

图灵社区：iTuring.cn

热线：(010)51095186转600

分类建议 计算机/软件开发

人民邮电出版社网址：www.ptpress.com.cn

TURING 2338354 www.ituring.com.cn 来社区 领银子

ISBN 978-7-115-38045-6

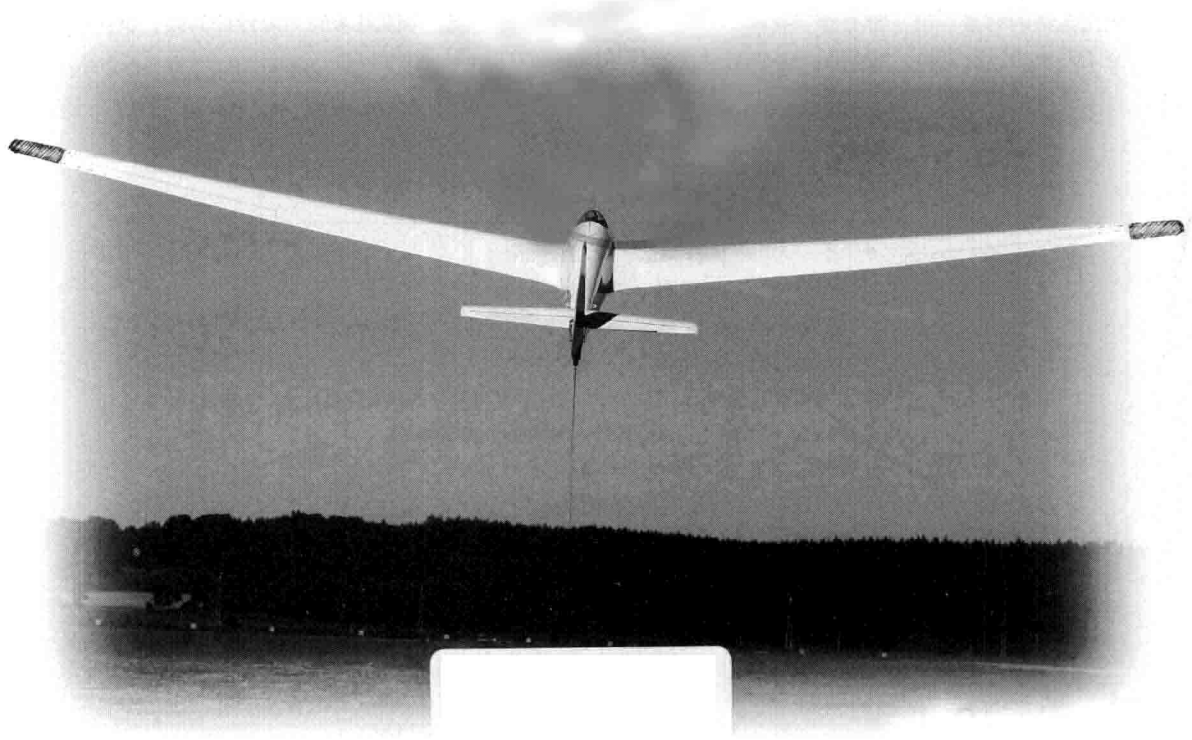
ISBN 978-7-115-38045-6

定价：49.00元

TURING 图灵程序设计丛书

发布！软件的设计与部署

【美】Michael T. Nygard 著　涂鸣 译



人民邮电出版社
北　京

图书在版编目（CIP）数据

发布！软件的设计与部署 /（美）尼加德(Nygard,M.T.）著；涂鸣译. -- 北京：人民邮电出版社，2015.2
（图灵程序设计丛书）
ISBN 978-7-115-38045-6

Ⅰ. ①发… Ⅱ. ①尼… ②涂… Ⅲ. ①软件设计 Ⅳ. ①TP311.5

中国版本图书馆CIP数据核字(2014)第300320号

内 容 提 要

作者根据自己的亲身经历和某些大型企业的案例，讲述了如何创建高稳定性的软件系统，分析设计和实现中导致系统出现问题的原因。本书分为四个部分，每部分内容都由一个研究案例引出。第一部分介绍了如何保证系统的生存，即维护系统正常运行。第二部分介绍了如何衡量系统的容量，以及如何随时间来优化系统的容量。第三部分讲述了架构师在为数据中心构建软件时应该思考的一般设计问题。第四部分讨论了系统的运行寿命，将其作为整个信息生态环境的一部分。

本书适合面向企业级软件的架构师、设计师和开发人员阅读参考。

◆ 著 [美] Michael T.Nygard
译 涂 鸣
责任编辑 朱 巍
责任印制 杨林杰

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：14.75
字数：349千字 2015年2月第1版
印数：1-3 000册 2015年2月北京第1次印刷
著作权合同登记号 图字：01-2011-1377号

定价：49.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第0021号

版权声明

Copyright © 2007 Michael T.Nygard. Original English language edition, entitled *Release It!: Design and Deploy Production-Ready Software.*

Simplified Chinese-language edition copyright © 2015 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由The Pragmatic Programmers, LLC.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

前　言

你为了自己的项目勤奋工作了一年多。最终，所有的功能似乎都已经完成，其中的大部分还进行了单元测试。你如释重负，终于大功告成了。

真是这样吗？

“功能完成”是否真的就意味着“产品就绪”了？系统确实可以部署了吗？操作员可以运行它，并独立面对大量实际用户了吗？你是否会有不祥预感，觉得半夜就会响起应急电话？其实，功能完成之外，还有很多开发工作要做。

通常，项目团队的目标是通过QA测试，而不是保障产品（注意这里说的是产品）的生命周期。也就是说，你的大部分工作是为了通过测试。测试，即便是敏捷、注重实效的自动化测试，也并不足以证明软件已经可以进行实际应用了。狂热的用户、全球涌来的超大流量，还有你闻所未闻的病毒制造者，这些残酷的现实都大大超出了测试时的预期。

为了确保软件能够应对自如，你的确需要做好充分的准备。在这本书里，我会为你展示问题所在，并帮助你了解如何解决它们。开始讲解之前，我会先讨论一些普遍存在的误解。

首先，你得接受这样的事实：尽管你的计划很棒，但是糟糕的状况仍然会出现。当然，如果可能，能够防范这些问题总是好的。不过最要命的情况就是，你以为自己总能预料并排除所有可能发生的错误事件。正确的做法应该是，对于自己确实能够预料到的问题采取预防措施，同时要确保整个系统能从那些无法预料的严重灾难中恢复过来。

其次，你要了解，发布1.0版并不是项目开发的终点，反而是系统自身生命周期的起点。这就好比刚长大成人的孩子第一次离开父母。你肯定不希望看到已经成年的孩子又搬回来和你一起住，特别是他们还带着另一半，加上四个孩子、两条狗和一只澳洲鹦鹉。

同样，你在项目开发过程中所做的决策将会极大影响你在1.0版发布之后的生活质量。如果你未能针对产品的环境来设计系统，那么你的生活会在项目发布后充满“刺激”，当然，那绝不会是惊喜。在本书中，你将会看到一些至关重要的设计折中，并且了解如何更明智地权衡利弊。

最后，尽管我们都热爱技术、看上去很炫的新技巧以及那些超酷的系统，但是你必须明白，这些都不重要。在现实的商业社会中（正是这个社会来为我们的系统买单），一切都归结到金钱。我

们打造的系统要花钱。为此，无论是开源还是节流，我们必须确保这些系统能够生钱。其他工作也要花钱，即便停工时也一样。所以，低效的代码会增加运营开销，从而浪费大量金钱。要理解一个运行系统，你必须了解钱的走向。同时，要想在行业中站有一席之地，你得赚钱，至少不能赔钱。

我真的希望这本书能使这些情况有所改变，切实地帮助你和你的公司避免传统企业软件所面临的问题：巨大损耗和过度支出。

谁来读这本书

本书面向企业级软件的架构师、设计师和开发人员，这些软件包括网站、网络服务、EAI系统等。在我看来，企业级意味着软件必须可用，否则公司就会赔钱。这些软件可以是通过销售直接获得收入的电子商务系统，也可以是雇员用于工作的重要内部系统。如果有人因为你设计的软件不能工作，不得不回家闲呆着，那么你需要好好读读这本书。

本书结构

本书分为四个部分，每部分内容都由一个研究案例引出。第一部分介绍如何保证系统的生存，即维护系统正常运行。尽管分布式系统通过冗余来保证可靠性，但是它更可能是以“两个8”而不是让人梦寐以求的“5个9”来展示其有效性[①]。相对任何其他考量，稳定性是必要的先决条件。试想，如果你的系统每天都会宕机，那么还有谁会关心这个系统的未来呢。在这种情形下，短期修复以及短期的思考将是主要手段。没有稳定性，未来就面临生存问题，因此你一开始就得找到方法，确保自己拥有一个可运转的稳定的基础系统。

一旦做到了稳定性，下一步要思考的就是容量。在第二部分，你会看到如何衡量系统的容量，了解容量的真正含义，并学习如何随时间来优化系统的容量。我会告诉你各种模式与反模式，借此诠释好的设计和坏的设计，以及它们如何对系统的容量产生显著的效果，当然也会影响半夜响起的应急电话的次数。

在第三部分，你会了解架构师在为数据中心构建软件时应该思考的一般设计问题。在过去的十年里，硬件和基础设施的设计已经发生了巨大的变化。例如，过去罕见的多宿主实现已经非常普遍了。网络变得日益复杂，它们分层设计并且更加智能。存储区域网络司空见惯。要让软件能够在数据中心里更加流畅地运行，软件设计必须考虑并利用这些变化。

在第四部分，你需要考察系统的运行寿命，将其作为整个信息生态环境的一部分。很多产品系统就像薛定谔的猫[②]，被锁在一只盒子中而无法观察其真实的状态。这不利于构筑一个健康的

① 即88%的正常运行时间，而非99.999%的正常运行时间。

② 薛定谔的猫是奥地利物理学家埃尔温·薛定谔为证明量子力学在宏观条件下的不完备性而提出的一个思想实验。——编者注

生态系统。缺乏信息，就不可能做出深思熟虑的改进[①]。第17章讨论了需要从开发中的系统学到的动机、技术和过程（这是你唯一可能获得某些经验的地方）。一旦系统的健康状况、性能和特性被暴露，你就可以针对这些信息采取行动。事实上，别无选择，你必须根据新的认识采取行动。不过，说来容易做来难，在第18章，你会看到变化带来的障碍，以及减少并克服那些障碍的方法。

关于案例

我加入了一些研究案例，用以阐述本书的主题。这些案例取自我所了解的真实事件及真正的系统故障。这些故障的代价非常高，让涉及其间的人感到非常尴尬。因此，为了保护这些公司和人，我故意模糊了一些信息。我修改了系统、类和方法的名称，但只改变了一些不重要的细节。在每个案例中，我都保留了行业、事件发生顺序、故障模式、错误传播和结果。这些故障的损失也没有夸大。这些案例都来自真实的公司，都涉及一大笔钱。我保留了这些数字，以强调这份材料的重要性。系统一旦发生故障，很可能就会损失一大笔钱。

致谢

这本书产生于我为对象技术用户组[②]所做的一次演讲。因此，感谢Kyle Larson和Clyde Cutting，他们推荐并认可了我的演讲。Poppendieck夫妇（Tom和Mary）是两本“精益软件开发”[③]图书的作者，他们给了我极大的鼓励，说服我把这个演讲内容整理成书。我还要特别感谢我的朋友和同事Dion Stewart，他不断地为这本书的初稿提供非常宝贵的反馈。

当然，还要感谢我的妻子和女儿们。我的小女儿出生后的一半时光都在看着我为此工作。她们如此耐心，陪伴我度过了无数个写书的周末。Marie、Anne、Elizabeth、Laura和Sarah，谢谢你们！

① 胡乱猜测可能碰巧取得改进，但更有可能增加而不是减少不确定性。

② 参见http://www.otug.org。

③ 参见*Lean Software Development*[PP03]和*Implementing Lean Software Development*[MP06]。

TURING 图灵教育

延展阅读



本书简明实用、见解深刻，总结了高效程序员在开发过程中的45个个人习惯、思想观念和方法，有助于开发人员在开发进程、编码工作、开发者态度、项目和团队管理以及持续学习这5个方面积极修炼。通过学习这些内容，养成这些好的习惯，你可以极大地提升自己的编程实力，更快速、更可靠地交付更高质量的软件，从而成为真正的高效程序员。

书名：高效程序员的45个习惯：敏捷开发修炼之道（修订版）
书号：978-7-115-37036-5
定价：45.00元



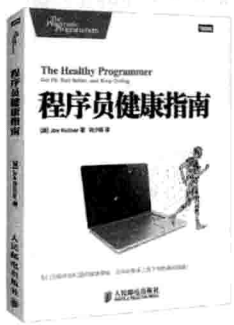
本书从认知科学、神经学、学习理论和行为理论角度，深入探讨了如何才能具备优秀的学习能力和思考能力，阐述了成为一名专家级程序员的关键要素，具体包括：大脑运行机制简介，如何正确使用和调试大脑，改进学习能力的具体技巧，如何通过自我引导积累经验，控制注意力的方法。为了让读者加深印象，作者还特别设立了一个“实践单元”，其中包括具体的练习和实验，旨在让读者真正掌握所学内容。

书名：程序员思维修炼（修订版）
书号：978-7-115-37493-6
定价：49.00元



作者多年来帮助许多高科技公司成功地解决了各种有关产品开发管理的棘手问题，本书正是她宝贵实战经验的提炼。书中从应对实际风险的角度出发，讲述了从项目启动、项目规划到项目结束的整个管理流程；展示了作者的思考过程，从评估项目背景，选择生命周期，直到为项目建立清晰的条件；同时穿插了丰富的提示和真实案例，介绍了可能遇到的常见问题。这些真知灼见不仅适用于软件项目管理，同样适用于其他产品的开发项目。

书名：项目管理修炼之道（修订版）
书号：978-7-115-36978-9
定价：59.00元



众所周知，程序员工作压力大、无暇照顾自己，去健身房更是奢侈行为。颈椎病、手腕疼痛、干眼症等问题困扰着几乎所有程序员和长期伏案的工作人员。在久坐不动的室内工作中，在没有大块时间去健身的客观条件下，如何保持身体健康？本书在科学研究和实践的基础上，总结了一些简单易行的健康指导，帮助读者从周身病痛、病弱且极有可能脾气暴躁的黑客，蜕变为快乐、高效的程序员。

书名：程序员健康指南
书号：978-7-115-36716-7
定价：39.00元

目　录

第 1 章　引言 ……1

1.1　瞄准正确的目标 ……1
1.2　使用决断力 ……2
1.3　生活的质量 ……3
1.4　挑战的范围 ……3
1.5　随手一松就是一百万 ……3
1.6　务实的架构 ……4

第一部分　稳定性

第 2 章　案例研究：航空系统宕机的异常 ……8

2.1　事故 ……9
2.2　结果 ……12
2.3　事后调查 ……12
2.4　确凿的证据 ……15
2.5　一点预防 ……17

第 3 章　稳定性概述 ……19

3.1　定义稳定性 ……20
3.2　故障模式 ……22
3.3　裂痕扩散 ……22
3.4　故障链 ……23
3.5　模式与反模式 ……24

第 4 章　稳定性反模式 ……26

4.1　集成点 ……27
4.2　连锁反应 ……37
4.3　连锁故障 ……40
4.4　用户 ……42
4.5　阻塞的线程 ……50
4.6　自我否定攻击 ……55
4.7　尺度效应 ……57
4.8　不平衡的容量 ……60
4.9　慢响应 ……63
4.10　SLA 倒置 ……64
4.11　无边界结果集 ……67

第 5 章　稳定性模式 ……70

5.1　使用超时 ……70
5.2　断路器 ……73
5.3　隔板 ……75
5.4　稳定状态 ……78
5.5　快速失效 ……83
5.6　握手 ……85
5.7　测试装置 ……87
5.8　去耦合中间件 ……90

第 6 章　稳定性总结 ……93

第二部分　容量

第 7 章　案例研究：被客户压迫 ……96

7.1　发布倒计时 ……96
7.2　瞄准 QA ……97
7.3　负载测试 ……99
7.4　被大量会话所杀 ……101
7.5　测试的鸿沟 ……102
7.6　后果 ……103

第 8 章　容量概述 ……105

8.1　定义容量 ……105
8.2　约束 ……106

8.3 关联 …… 107
8.4 可扩展性 …… 107
8.5 容量的神话 …… 108
8.6 总结 …… 114

第 9 章 容量反模式 …… 115

9.1 资源池竞争 …… 115
9.2 泛滥的 JSP 碎片 …… 118
9.3 AJAX 过度之伤 …… 119
9.4 驻留过久的会话 …… 121
9.5 HTML 中浪费的空间 …… 122
9.6 刷新按钮 …… 125
9.7 手工的 SQL 语句 …… 126
9.8 数据库富营养化 …… 128
9.9 集成点延迟 …… 130
9.10 Cookie 怪兽 …… 131
9.11 总结 …… 133

第 10 章 容量模式 …… 134

10.1 连接池 …… 134
10.2 谨慎使用缓存 …… 136
10.3 预计算容量 …… 137
10.4 调整垃圾回收器 …… 140
10.5 总结 …… 142

第三部分 一般设计问题

第 11 章 网络连接 …… 144

11.1 多宿主服务器 …… 144
11.2 路由 …… 146
11.3 虚拟 IP 地址 …… 146

第 12 章 安全 …… 149

12.1 最少特权原则 …… 149
12.2 配置的密码 …… 150

第 13 章 可用性 …… 151

13.1 收集可用性需求 …… 151
13.2 记录可用性需求 …… 152
13.3 负载均衡 …… 153
13.4 集群 …… 157

第 14 章 管理 …… 159

14.1 “测试和产品匹配吗？” …… 159
14.2 配置文件 …… 161
14.3 启动和关闭 …… 163
14.4 管理接口 …… 164

第 15 章 设计总结 …… 165

第四部分 运营

第 16 章 案例研究：惊人的宇宙 …… 168

16.1 旺季 …… 168
16.2 婴儿的第一个圣诞 …… 169
16.3 切脉 …… 169
16.4 感恩节 …… 170
16.5 黑色星期五 …… 170
16.6 重要的信号 …… 172
16.7 诊断测试 …… 172
16.8 专家打来电话 …… 173
16.9 比较解救方案 …… 174
16.10 条件是否会响应处理 …… 175
16.11 收尾 …… 176

第 17 章 透明度 …… 177

17.1 视角 …… 178
17.2 透明度设计 …… 184
17.3 使用各种技术 …… 184
17.4 日志 …… 185
17.5 监控系统 …… 190
17.6 法律上及事实上的标准 …… 194
17.7 操作数据库 …… 201
17.8 支持流程 …… 205
17.9 总结 …… 208

第 18 章 适应 …… 209

18.1 与时俱进 …… 209
18.2 适应性的软件设计 …… 210
18.3 适应性的企业架构 …… 215
18.4 发布应无害 …… 220
18.5 总结 …… 224

参考书目 …… 226

第1章 引 言

学校或社会开设的软件设计课程极其不完整，它只谈了系统该做什么，而没有涉及系统不该做什么。系统不应该崩溃、宕机、丢失数据、侵犯隐私、损失金钱、损害公司，或者导致客户流失。

在本书中，我们会探讨为这个混乱、扭曲的真实世界架构、设计和构建软件，特别是构建分布式系统的方法。为应对那些行事疯狂、难以预测的用户，我们要做好准备。软件从发布的那一刻起就会遭受攻击，它需要抵抗台风般的快闪族、Slashdotting，抑或 Fark 或者 Digg[①]上的链接。我们将仔细研究无法通过测试的软件，找到方法确保软件能经受现实世界的考验。

现今的软件设计就像 20 世纪 90 年代早期的汽车设计，完全脱离现实世界。那时候，汽车只是在凉爽舒适的实验室中设计，从模型和 CAD 系统看，它们都很棒。在几个巨大风扇前，曲线完美的汽车闪闪发光，轰隆着冲破层层气流。这些设计者们生活在这样平静的空间里，进行着优雅、复杂、精巧，但也脆弱、不能让人满意并且最终短命的设计。大多数的软件架构和设计也同样在纯净、与世隔绝的环境中进行。

你想拥有一辆为现实世界设计的汽车，你希望这位设计者懂得：机油要在汽车每行驶 3000 英里后更换；轮胎胎面磨损至 1/16 英寸时，性能必须与开始时一样好；在你一手拿着腌肉蛋汉堡，一手打着电话时，你保不齐就会猛踩刹车。

1.1 瞄准正确的目标

大多数软件是为研发实验室或者质量保障（QA）部门的测试人员设计的。设计与构建软件

① 快闪族是指一群素昧平生的人通过网络、手机短信等途径事先约定活动主题、时间和地点，然后一起做出夸张举动，这种活动通常短得令旁人来不及反应。Slashdotting 来源于 Slashdot.org 网站，该网站文章中的网址链接有可能在一瞬间被点击上千次，甚至上万次，造成被链接网站负荷不了，产生画面宕掉或带宽被完全占用的现象。Fark 和 Digg 的网址分别是 fark.com 和 digg.com。——译者注

的目标是为了通过测试，比如“必须输入客户的姓和名，中间名可选择输入”。它的目的是为了能在 QA 的人造场景，而非真实的产品世界中使用。

当系统通过了 QA 这一关，我就能自信满满地说它已经能够作为产品发布了吗？仅仅通过了 QA 测试，并不能让我了解，系统在今后的 3 ~ 10 年中是否能稳定工作。它可能是软件中的丰田凯美瑞，连续运行上千个小时；也可能是雪佛兰织女星（该车的前端曾经在公司自己的测试跑道上解体）或者福特平托，会在正常的碰撞测试时发生爆炸。几天或几个星期的 QA 测试并不能说明未来数年会发生什么。

长久以来，制造业的产品设计者一直在追求“可制造性设计”，这种设计产品的工程方法能够保证低成本、高质量地制造产品。在此之前，产品设计者与制造者自行其是，各自为战。设计部门“隔墙”丢给生产部门的方案，竟然包含了不合格的螺钉、易混淆的零件，以及需要用现成部件定制的零件。质量低劣、制造成本高昂自然是不可避免的。

这听来熟悉吗？今天，软件业也处在这样的状态。我们总是无法按期完成新的系统，因为要不断接听电话，为已经被我们甩掉的上一个未完成项目提供技术支持。我们以为“可制造性设计”就是“为生产而设计”，但其实我们不是为制造者设计软件，而是要将完成的软件交付 IT 运营部门。我们需要设计出一个个独立的软件系统，以及相互依赖的多个系统所构成的整个生态系统，以此带来低成本和高质量的 IT 运营。

1.2 使用决断力

早期决策对系统的最终形态影响最大。最初的决策最难以更改。这些关于系统边界和子系统划分的早期决策，渗入到了团队结构、经费分配、程序管理结构甚至工时表中。团队分配是架构的第一稿（参见 7.2 节知识框）。极具讽刺意味的是，这些早期决策也是在信息获取最少的情况下做出的。这就是说，你的团队在这个时候对软件的最终结构最无知，但你必须要做出某些不可更改的决策。

即使是“敏捷”项目[①]，也最好有远见地做出决策。为了选择最健壮的设计，设计者预测未来的时候似乎必须有决断力。由于不同的选择通常会有相似的实现成本，却带来完全不同的生命周期开销，因而考虑每种决策对可用性、能力和灵活性的影响就显得格外重要。我将结合较好和较差方法的具体例子，阐述各种设计选择的效果 。这些例子都来自于我接手过的真实系统，大多数都曾让我牺牲了宝贵的睡眠。

① 我声明，我是敏捷方法的强烈推崇者。它强调及早发布并持续改进，意味着软件可以尽快投入使用。因为软件投入使用是唯一得知软件如何响应真实世界刺激的途径，我提倡不管使用什么方法，都要尽早开始这个学习过程。

1.3 生活的质量

发布 1.0 版是软件生命的开始，而不是项目的结束。1.0 版发布后，你的生活质量取决于你在这个重要里程碑之前所做的选择。

不管你是按小时收费的技术支持工程师，还是雇用这些人的雇主，都需要知道你要维护的是一个经得起严格测试、坚不可摧，并能推进你的事业向前发展的汽车，而不是一个只在卖场摆放的花瓶。

1.4 挑战的范围

“软件危机”已经超过 30 年了。就金主而言，软件耗资依然太大（参见 *Why Does Software Cost So Much?* [DeM95]）。就目标捐助者而言，即便进度表是以月而非年来衡量的，软件的开发周期依然很长。很明显，过去 30 年设想的生产效率提高只是一场幻影。

这些术语来自于敏捷社区。金主（gold owner）是指为软件付费的人。目标捐助者（goal donor）是指你正在满足其需求的人。他们一般都不是同一个人。

另一方面，我们获得的某些实际的生产效率提高方法，可能被用来攻克更大的问题，而不是快速廉价地开发同样的软件。在过去的十年里，系统的规模呈指数级膨胀。

在安逸懒散的客户端/服务器系统时代，系统的用户基数是以十或百来衡量的，并发用户数顶多也就几十个。可现在，资助人要求系统要能承受“25 000 个并发用户”和“每天 400 万独立访问者”。

正常运行时间的要求也提高了。然而，著名的“5 个 9”（99.999%）运行时间曾经是大型机及其维护者给出的承诺，而现在，用户希望普通的商业网站也能够提供 24×7×365 的服务[①]。显然，从今天构建的软件系统的规模来说，我们已经向前迈了一大步，但是系统范围和规模的增大也导致新的系统崩溃方式层出不穷、环境更加恶劣，以及用户对系统缺陷的容忍度降低。

快速廉价地构建软件，对用户更友好，操作更简便，这些新的挑战要求我们持续地改进软件架构和设计方法。对小型网站来说可行的设计方法，对有上千用户的事务型分布式系统是不堪一用的，我们将会分析一些恐怖故障的案例。

1.5 随手一松就是一百万

项目的成功、股票期权或利润分红、公司的生存，甚至你的工作，所有这些都岌岌可危。为

① 这个短语总是在困扰我。作为一名工程师，我希望它要么是“24×365”，要么是“24×7×52”。

QA测试而构建的系统，通常在操作成本、宕机时间、软件维护等方面，都需要很大的持续支出，它们根本没法盈利，更别说有什么净现金收益了（这是指系统减去了构建成本之后的利润）。这些系统的可用性低，直接导致了收入减少，有时甚至因损害品牌形象而产生更大的间接损失。对于我的很多客户来说，宕机的直接损失每小时在10万美元以上。

在一年里，98%的正常运行时间和99.99%的正常运行时间，二者在收益上的差别将超过1700万美元[①]。请想象一下，仅仅通过更好的设计就能为账户添加1700万美元！

在繁忙紧张的项目开发期间，你很可能会决定以牺牲运营成本来优化开发成本。这只是在项目团队有固定预算和交付时间的情况下才有意义。然而，这对为软件付费的组织来说，是个错误的选择。系统的运营时间要远远多于开发时间，至少对那些还没有被取消或废弃的系统来说是如此。靠承担经常性运营成本来避免一次性成本，是没有意义的。事实上，通过一次性投入减少经常性运营成本的经济效益更大。如果为了避免版本发布期间的宕机，你愿意在一个自动构建和发布版本的系统上花费5000美元，那么公司将会少损失20万美元[②]。我相信，大多数CFO都会因这4000%的投资回报率而批准这笔经费的。

系统设计与架构上的决策也是财务决策，制定这些决策都必须着眼于实现成本和宕机成本。这种技术和财务相融合的观点，是本书反复探究的一个最重要的主题。

> 不要为了避免一次性开发成本，而以经常性运营成本为代价。

1.6 务实的架构

两种不同的活动都归结到了架构这一术语。一种类型的架构追求高度抽象，以做到更灵活的跨平台移植，并更少地关联硬件、网络、电子和光子等复杂细节。这种方法的极端形式会造就一座“象牙塔”——一间Kubrickesque式的整洁房间，盒子和箭矢装饰着每面墙，超然不俗的大师居住在这里。命令从象牙塔发出，降临在劳苦的程序员身上。“使用EJB的CMP完成数据库操作！”“所有UI使用JSF构建！”“所有过去、现在和将来的数据都要保存在Oracle数据库中！”如果你曾经为了遵守这些“公司标准”而在编码时咬紧牙关（用其他的技术完成会容易十倍），那你就是象牙塔架构师的受害者。我敢保证，一个不愿倾听团队成员意见的架构师，也一定不愿倾听用户的声音。你已经看到结果了：当系统崩溃时，用户们欢欣鼓舞，因为他们可以暂时不受这个烂系统的折磨了。

① 对于一级零售商，平均宕机成本在每小时10万美元。

② 这是假设每次发布需要1万美元（劳动力加上计划宕机的成本），每年有4次发布，计划5年。大多数公司都希望每年有超过4次发布，但是我比较保守。

另一类架构则是架构师与程序员们同心协力，甚至成为其中一员。这些架构师会毫不犹豫地剥去抽象或者摒弃那些不适合的部分。务实的架构师更喜欢讨论诸如内存使用、CPU需求、带宽要求，以及超线程和CPU绑定的优缺点等问题。

象牙塔架构师非常喜欢一种完美的终极愿景，而务实的架构师会不断地思考变化的原因。“我们该如何不重启系统而完成部署？”“我们需要收集什么数据，以及该如何分析它们？”“系统的哪部分最需要改进？”象牙塔架构一旦完成，系统就不容许任何改进，每部分都将完美地适应其角色。相反，务实架构师构建的每个组件都可以轻松地应对当前的问题，他们知道根据系统压力随时间变化的方式，哪些模块将被替换掉。

如果你已经是一个务实的架构师，书里为你准备了各种强力“弹药”。如果你是一个象牙塔架构师，并且还在读这本书，那么本书可能会“怂恿”你降低一些抽象层次，来重新思考软件、硬件和用户间的重要交叉点：活在产品中。这样，在产品最终发布的时候，你、用户以及公司都会高兴很多！

Part 1

第一部分

稳定性

本部分内容

- 第 2 章　案例研究：航空系统宕机的异常
- 第 3 章　稳定性概述
- 第 4 章　稳定性反模式
- 第 5 章　稳定性模式
- 第 6 章　稳定性总结

第2章
案例研究：航空系统宕机的异常

你是否注意过一些起于非常小的事情而后演变成了大问题的事件？微小的编程错误就像雪球滚下山坡，随着动力的增加，问题的规模也越来越大。一家大型航空公司就经历过这样的事件，该事件最终使数千名乘客滞留，公司损失数十万美元。下面说说这个事故的经过。

事故肇始于对核心设施（CF，Core Facilities）数据库集群进行的一次例行失效备援操作。[①]该航空公司正试图将系统转向面向服务的架构，以期提高复用性，减少开发时间，降低运营成本。当时，CF 系统还处于第一代。CF 团队计划按功能分阶段上线。这是一个合理的计划，并且听起来也很熟悉——多数大公司都在进行类似的项目。

CF 负责处理航班的搜索，这个服务在航班系统中非常普遍。给定日期、时间、城市、机场代号、航班号，或者任意的条件组合，CF 都能查找并返回一组详细的航班信息。事故发生的时候，自助值机亭、交互式语音应答（IVR，Interactive Voice Response）系统和“渠道合作”应用都已经更新，在使用 CF 系统。渠道合作应用会为大型旅游预订网站生成输入数据。IVR 和自助值机亭用来帮助乘客登机，用土话说就是“一个萝卜一个坑”。开发时间表计划发布新版本的登机程序和呼叫中心系统，以过渡到 CF 系统，但实际上并未推出，你会看到，这反而是件好事。

交互式语音应答：可怕的电话菜单系统。

CF 系统的架构师完全意识到了这次系统升级有多么重要。他们的目标就是高可用性。系统运行在一个 J2EE 应用服务器集群上，使用 Oracle 9*i* 数据库存储冗余数据。所有的数据都存储在一个大型外置 RAID 阵列上，使用场外磁带每天备份两次，第二个机架上的磁盘副本保存五分钟前的数据。

Oracle 数据库服务器每次都运行在集群的一个节点上，Veritas 集群服务器用来控制数据库服务器、分配虚拟 IP 地址，以及从 RAID 阵列上安装和卸载文件系统。在最前端，一对冗余的硬

① 一如既往，我更改了所有的人名、地名和日期，以保护参与的人和公司的隐私。

件负载平衡器负责将输入的流量定向到一台应用服务器上去。调用自助值机亭和 IVR 系统等应用程序，会连接到前端的虚拟 IP 地址上去。到目前为止，一切都正常。

如果你做过网站或网络服务的工作，图 2-1 很可能会看起来很熟悉。这是一个非常常见的高可用性架构，并且设计良好。CF 系统不存在任何常见的单点故障问题。CPU、风扇、硬盘、网卡、电源和网络交换机，每一个硬件都有冗余。服务器甚至能分割成多个机架，以防止某个机架意外被毁。实际上，当发生火灾、水灾、爆炸，甚至流星碰撞时，远在 50 公里外的第二集群可以马上接管所有工作。

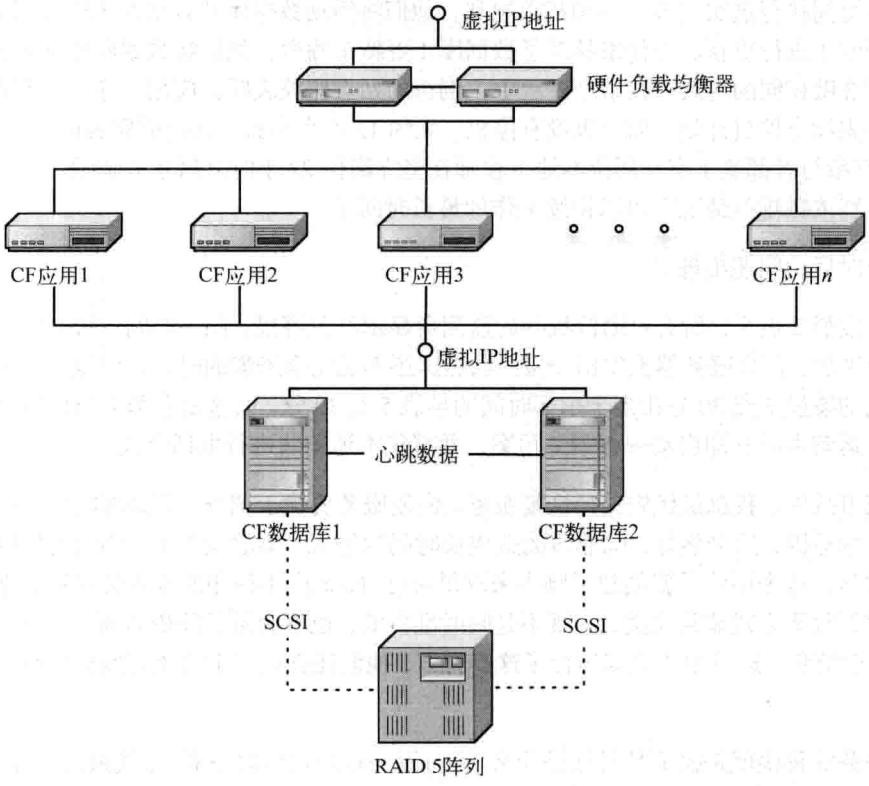


图 2-1 CF 系统开发架构图

2.1 事故

如同我的多数大客户一样，本地的工程师团队专职负责运营这家航空公司的基础设施。实际上，事故发生时，这个团队已经负责此项工作三年多了。出事的那天晚上，本地工程师刚刚执行了一次人工的数据库失效备援，将 CF 数据库 1 转移到 CF 数据库 2（参见图 2-1）。他们使用 Veritas

系统将活动数据库从一台主机迁移到了另一台主机。这样，他们就可以对第一台主机进行例行的维护工作。完全是例行的，这个流程他们过去已经做了几十遍。

Veritas 集群服务器负责执行失效备援。在一分钟内，它能关闭数据库 1 上的 Oracle 服务器，卸载 RAID 阵列上的文件系统，并重新装入到数据库 2，启动数据库 2 上的 Oracle 服务器，然后再把虚拟 IP 地址分配给数据库 2。应用服务器觉察不出任何变化，因为它们被配置为只连接到这个虚拟 IP 地址。

这个客户计划在太平洋时间周四晚上 11 点左右进行这次特别更改。一位本地团队的工程师与运营中心协同执行这次更改。一切都很顺利，他们将活动数据库从数据库 1 迁移到数据库 2，然后对数据库 1 进行更新。在仔细核对了数据库 1 更新正确后，他们将数据库迁移回数据库 1，并对数据库 2 做相同的更新。在整个过程中，例行的站点监控表明，应用程序一直都正常可用。这次变化事先没有停机计划，而且也没有停机。大约 12 点半左右，这次更新被记录为“已完成，成功”，所有参与者都签了字。那位本地工程师在坚守岗位 22 小时之后才去睡觉。毕竟，这是在喝了两大杯特浓咖啡的情况下可以持续工作的最长时间了。

两个小时后，问题出现了。

大约在凌晨 2 点半，所有自助值机亭的监测台显示红色警报，同一时间，所有地方都停止了服务。几分钟后，IVR 服务器也发出警报。虽然这还不是完全恐慌的时刻，但也差不多了，因为太平洋时间的凌晨 2 点 30 分相当于东部时间的早晨 5 点 30 分，而这是东部沿海地区乘客登机的高峰时间。运营中心立即启动一级紧急预案，并召集本地团队进行电话会议。

在任何事故中，我总是优先选择恢复服务，恢复服务要优于调查。若是能收集一些数据以便将来分析事故原因，这当然好，除非事故造成长时间的停机。事故发生了，临时抱佛脚肯定行不通。幸运的是，这个团队早就创建了脚本来取得所有 Java 应用程序的线程转储和数据库快照。这种自动数据收集方式非常完美，它既不是临时准备的，也不会延长停机时间，还有助于事后分析。按照这个流程，运营中心立即运行了这些脚本，他们还尝试了重启该自助检票系统的一台应用服务器。

恢复服务的秘诀就是搞清楚目标是什么。通过逐一重启每台服务器，系统最终总能全部重启。这几乎总是有效的，但太耗时。大多数时候，你是能够找到罪魁祸首的，就像医生诊断疾病一样。虽然你能够医治病人的每个已知疾病，但那将是一个痛苦、昂贵和缓慢的过程。相反，你可以看看病人表现的症状来弄清到底该治什么病。难的是许多症状并不明显，当然，有时候某些症状会让医生直接抓住根本问题，但这并不常见。多数时候，你了解的症状（比如发烧）并不能说明什么，有数以百计的疾病都能引起发烧。为了在各种可能的原因中辨别出真正的原因，你需要通过测试或观察得到更多的信息。

在这个案例中，团队面临的是两套完全挂起并独立的应用程序，它们几乎同时挂起，微小的

时间差异可能只是来自检票机和 IVR 应用程序使用的不同监控工具的检测延时。显而易见，大家首先想到的就是两套应用程序都依赖了某个存在缺陷的第三实体。正如图 2-2 所示，那就是 CF，它是检票机和 IVR 系统所共用的唯一组件。在问题出现前 3 个小时，CF 进行了一次数据库失效备援，因此它被高度怀疑。但是，监测并没有报告 CF 存在故障。日志文件里也看不出任何问题，网址探测也是。结果是监控程序只是给出了一个状态页面，因而大家还是无从判断 CF 应用服务器的真实状况。后来，我们通过常规的渠道修正了这个错误，并做了记录。

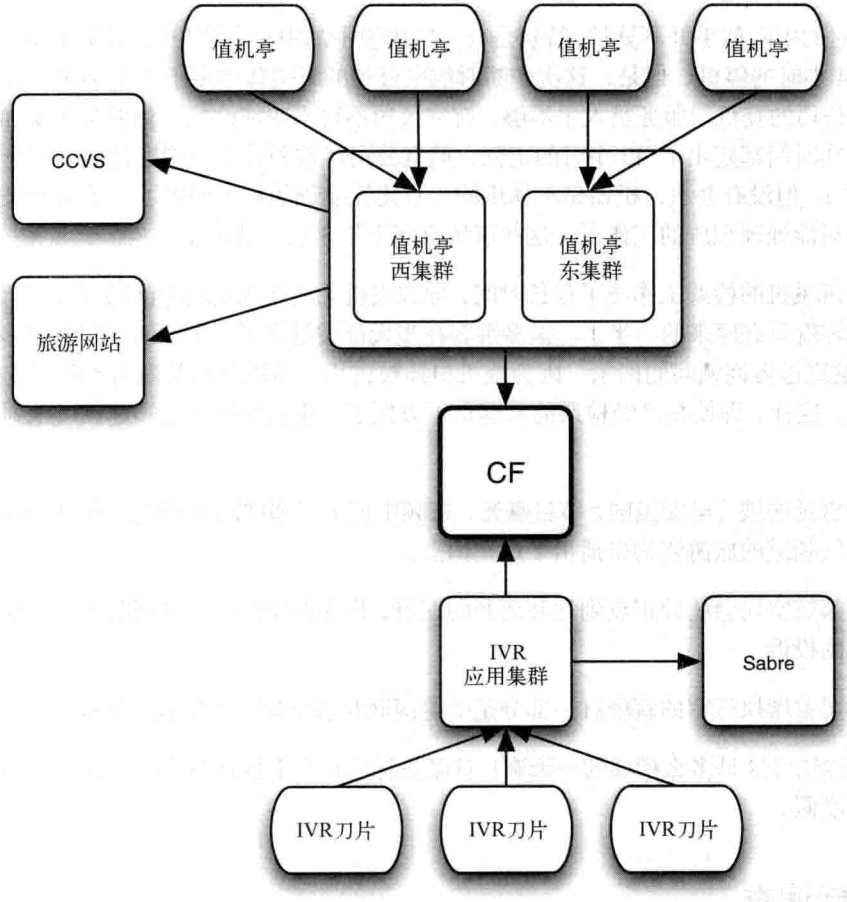


图 2-2 公共依赖关系

请记住，恢复服务是优先级最高的任务。这次中断快要接近 1 小时服务水平协议（SLA）的限制了，因此团队决定重启 CF 的每个应用服务器。当他们重启第一台服务器时，IVR 系统就开始恢复了。重启了所有 CF 后，IVR 系统都能正常运行，但是检票机仍然不能工作。根据直觉，首席工程师决定重启检票机自己的应用服务器，这确实起了作用，检票机和 IVR 系统显示版上

绿灯都亮了，显示运转正常。

服务水平协议：网络服务供应商和客户之间的合同，违反 SLA 通常会受到经济处罚。

这次事故持续了大约 3 小时，从太平洋时间晚上 11:30 到凌晨 2:30。

2.2 结果

3 个小时的中断似乎并不算长，特别是与一些传说中的中断事故相比，比如 eBay 曾经在 1999 年发生了 24 小时的停机。但是，这次中断对航空公司的影响却远多于 3 个小时。启用旧系统检票后，航空公司的登机门服务员人手不够，航空公司不得不叫回已经下班的员工来加班。很多人在这周的工作时间都超出了 40 小时的定额，航空公司得按照劳工合同支付他们加班费（正常薪金的 1.5 倍）。但没有办法，机器是不休班的，只能让换班的员工回来了。等到换班的员工抵达现场，他们只能处理积压的工作了，这些事情直到下午 3 点才处理完。

由于早班飞机的检票工作花了很长时间，导致飞机无法在规定的时间离开登机口，若准点起飞，登机乘客就只有原来的一半了。很多乘客在当天都被延误了。星期四正好有很多“呆鸟”航班[①]，这些航班送咨询顾问们回家。因为登机口还被占用，所以抵达的航班不得不安排在其他空闲的登机口。这样，即使是已经检票的乘客也不方便了，他们要从原定的登机口跑到重新安排的登机口。

这次飞机延误被《早安美国》节目曝光，视频中记录了那些陷入困境的独自带着婴儿的可怜妈妈们。天气频道的旅游咨询也播出了这则消息。

美国联邦航空局会统计正点到达和离开的航班，作为航空公司年度报告的部分记录。他们还会统计客户的投诉。

航空公司首席执行官的薪酬有一部分是受美国联邦航空局的年度报告影响的。

所以，你知道这是多么糟糕的一天啊！首席执行官正在催促运营中心，找出是谁导致他不能去圣托马斯度假。

2.3 事后调查

太平洋时间上午 10 点半，也就是停机 8 小时以后，我们的客户代表汤姆[②]叫我做事后调查。因为故障是紧跟在数据库失效备援和维护之后发生的，所以调查也就围绕这些展开了。在运营方

① “呆鸟”航班是指往返于高新科技城市的直达航班，大部分旅客都是搞研究的呆子，因而得此名。——编者注

② 化名。

面，“发生其后，故因其所致”[①]在大多时候被证明是个好的着手点。虽然这并不总是正确的，但它提供了可以寻找线索的途径。实际上，当汤姆打电话给我时，他是要求我飞过去的，找出为什么数据库失效备援会导致这次停机。

在飞机上，我就开始用笔记本审查初始的事故报告。我的工作任务很简单：展开事后调查，回答下面这些问题。

- ❑ 是数据库失效备援导致了这次停机吗？如果不是，那是什么原因？
- ❑ 服务器集群的配置正确吗？
- ❑ 运营团队进行了正确的维护吗？
- ❑ 停机以前就有事故迹象了吗？
- ❑ 最重要的是，我们该如何确保下次不会发生这样的事了？

当然，我在现场还要向客户表明我们在慎重应对这次停机。不用说，我的调查也要减轻大家对本地团队粉饰这次事故的担心。当然，他们不会这样做，但是重大事故之后的管理认知和管理事故本身同样重要。

重大事故之后的管理认知和管理事故本身同样重要。

事故调查就像侦察一桩神秘的谋杀案。你有一系列的线索，有些可靠，比如停机时刻保留下来的服务器日志；有些不可靠，比如相关人对他们所见的描述。就如目击证人，人们会将猜测混于观察中，将假设陈述为事实。可以说，事故分析比侦破谋杀案更艰难，因为“尸体”已经消失了。没有“尸体”可供检验，因为这些服务器还在备份和运行。它们在导致事故时不管是处于何种状态，现在都不能再现了。故障可能会在当时的日志或监控数据中留下痕迹，但也可能没有，因此这些线索都很难追踪。

在阅读这些文件时，我对收集到的数据做了些记录。对于应用服务器，我需要日志文件、线程转储和配置文件；对于数据库服务器，我需要数据库和集群服务器的配置文件。我还对当前配置文件和当夜备份做了比较记录。备份工作在停机前就开始了，由此我能够知道在备份和我的调查之间是否有任何配置上的改变。换句话说，它会告诉我是否曾有人掩盖失误。

回到宾馆，我的身体疲惫不堪，此时已经过了午夜。我只想洗个淋浴，躺在床上，可是我被在线“监禁”了，客户经理要向我简要介绍相关进展。终于，在凌晨 1 点左右，汇报结束了。

当天上午，喝了些咖啡，我就钻研起数据库集群和 RAID 配置文件了。我在寻找一些集群常有的问题：没有足够的心跳数据，心跳数据通过传输业务流量的交换机，服务器设置为使用物理

① post hoc，ergo propter hoc，其字面意思是“在这个之后，因此这个就是原因”。这是常见的基于紧邻事件寻求原因的逻辑谬论。

IP 地址而不是虚拟地址，软件包之间的错误依赖关系，等等。当时，我并没有检查清单，这些都是我不止一次看到或听到的问题。我没有找到出错的地方，工程团队确实出色地维护了数据库集群，如同教科书一般。实际上，他们使用的某些脚本就直接来自 Veritas 的培训教材。

接下来，是检查应用服务器的配置了。本地工程师复制了检票机在故障期间的所有日志。我也能够从 CF 应用服务器上取得日志文件，停机时的日志文件也在，因为事故只是发生在头一天。更妙的是，所有日志文件都设置了线程转储。作为一个老 Java 程序员，我很愿意使用 Java 线程转储来调试应用程序挂起的故障。

有了线程转储，如果你知道如何阅读它，那么应用程序就一目了然了。即便你从未见过应用程序的源代码，也能推导出许多东西。你能够知道应用程序使用的第三方库，何种线程池，每个池有多少线程，应用程序使用哪种后台处理。查看每个线程栈中的类和方法，你甚至能说出应用程序使用的是什么协议。

没过多久，我就将问题定位在了 CF 上。检票机的应用服务器的线程转储，清楚地表明了我所期望的从事故期间的可见信息中了解到的东西。有 40 个专门处理检票机请求的线程，所有线程都阻塞在 `SocketInputStream.socketRead0()`这个方法上，这是 Java 的套接字库内部的一个原生方法。它们在徒劳地读取一个永远都不会到来的响应。

获取线程转储

当你向 Unix 系统发送信号 3（SIGQUIT）或者在 Windows 系统按下 Ctrl+Break 时，Java 应用程序都会转储 JVM 上每个线程的状态。

要在 Windows 系统上生成线程转储，必须以命令行方式运行 Java 程序。当然，如果想远程登录，就需要使用 VNC 或远程桌面。

在 Unix 系统上，可以使用 `kill` 命令发送这个信号：

```
kill -3 18835
```

关于线程转储的一个难题是：它们总来自“标准输出”。很多已经停用的启动脚本并不捕捉标准输出，或者发送到/dev/null 了。（例如，Gentoo Linux 系统上的 JBoss 包将 JBOSS_CONSOLE 默认设置到/dev/null。）由 Log4J 或 java.util.logging 产生的日志文件不会显示线程转储。你可能不得不操作应用服务器的启动脚本来获得线程转储。

下面是一小段 JBoss 3.2.5 的线程转储：

```
"http-0.0.0.0-8080-Processor25" daemon prio=1 tid=0x08a593f0 \
  nid=0x57ac runnable [a88f1000..a88f1ccc]
    at java.net.PlainSocketImpl.socketAccept(Native Method)
    at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:353)
    - locked <0xac5d3640> (a java.net.PlainSocketImpl)
```

```
        at java.net.ServerSocket.implAccept(ServerSocket.java:448)
        at java.net.ServerSocket.accept(ServerSocket.java:419)
        at org.apache.tomcat.util.net.DefaultServerSocketFactory.\
      acceptSocket(DefaultServerSocketFactory.java:60)
        at org.apache.tomcat.util.net.PoolTcpEndpoint.\
      acceptSocket(PoolTcpEndpoint.java:368)
        at org.apache.tomcat.util.net.TcpWorkerThread.\
      runIt(PoolTcpEndpoint.java:549)
        at org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.\
      run(ThreadPool.java:683)
        at java.lang.Thread.run(Thread.java:534)
    "http-0.0.0.0-8080-Processor24" daemon prio=1 tid=0x08a57c30 \
      nid=0x57ab in Object.wait() [a8972000..a8972ccc]
        at java.lang.Object.wait(Native Method)
        - waiting on <0xacede700> (a org.apache.tomcat.util.threads.\
      ThreadPool$ControlRunnable)
        at java.lang.Object.wait(Object.java:429)
        at org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.\
      run(ThreadPool.java:655)
        - locked <0xacede700> (a org.apache.tomcat.util.threads.\
      ThreadPool$ControlRunnable)
        at java.lang.Thread.run(Thread.java:534)
```

这里包含了比较丰富的内容。

这个片段显示了两个线程，每个都像 http-0.0.0.0-8080-ProcessorN 一样命名。25 号线程处于可运行状态，而 24 号则被阻塞在 Object.wait()。这段代码清楚地指明，这些是一个线程池的成员。那些在栈上命名为 ThreadPool$ControlRunnable()的类也可能是线索。

检票机应用服务器的线程转储也清楚地告诉了我所有 40 个线程调用的类和方法：FlightSearch.lookupByCity()。我很惊讶地看到，在栈上有些帧对 RMI 和 EJB 方法的引用较高。CF 总是被描述为“网络服务”。诚然，网络服务的定义在那时并不精确，但称无状态会话 Bean 是“网络服务”好像还是有些牵强。

RMI 给 EJB 提供远程过程调用。EJB 调用能够建立在 CORBA 或 RMI 两种传输上。虽然我喜欢 RMI 的编程模型，但由于调用不能超时退出，因此相当危险。结果，调用者面对远程服务器故障时就显得很脆弱。

2.4 确凿的证据

有一点，事后分析与停机本身的现象是相吻合的：CF 造成了 IVR 和检票机登入系统挂起。最大的悬疑是，CF 到底怎么了？

当我仔细查看 CF 的线程转储时，事情变得愈发明朗了。CF 的应用服务器使用了不同的线程

池来处理 EJB 调用和 HTTP 请求。这就是为什么 CF 总能对监控程序做出响应，即便是在故障发生时。HTTP 线程几乎完全闲置，这对 EJB 服务器很重要。另一方面，EJB 线程被完全用于处理 `FlightSearch.lookupByCity()`的调用。实际上，每个应用服务器的每个线程都被完全阻塞在同一行代码：尝试从资源池中获得数据库连接。

这只是旁证，而非确凿的证据。不过考虑到事故是在数据库的失效备援操作之后发生的，那么看起来我的方向是正确的。

接下来的部分有点冒险，我需要查看代码，但是运营中心无法访问源代码管理系统。只有那些二进制文件被部署到生产环境中，这通常是好的安全防范措施，但此刻很不方便。当我问及客户经理我们该如何访问源代码时，他很不情愿。由于事故的规模重大，你可以想象空气中弥漫着一股火药味儿，很多人正怒气冲冲地等着把肇事者揪出来。运营中心和开发部门的关系从来就不那么和气，而此时比平常更加紧张。每个人都在防守，谨防任何追究责任的矛头指向自己。

因此，在不能合法访问源代码的情况下，我做了自己唯一能做的事情，那就是把二进制文件反编译出来[①]。当我看到了那可疑的代码时，就知道了事故的确凿原因。这个特别的会话 bean 是 CF 实现的唯一功能。

```
package com.example.cf.flightsearch;

. . .

public class FlightSearch implements SessionBean {

  private MonitoredDataSource connectionPool;

  public List lookupByCity(. . .) throws SQLException, RemoteException {
    Connection conn = null;
    Statement stmt = null;

    try {
      conn = connectionPool.getConnection();
      stmt = conn.createStatement();

      // Do the lookup logic
      // return a list of results
    } finally {
      if (stmt != null) {
        stmt.close();
      }

      if (conn != null) {
        conn.close();
      }
    }
  }
}
```

① 我最喜欢的反编译 Java 代码工具仍然是 JAD。它快速而准确，虽然用在 Java 5 代码上时会磕磕绊绊的。

其实，乍一看，这种写法是合理的。使用 try...finally 表明了写代码的人清理资源的意图。而且，这样的清理代码经常出现在市售 Java 图书上。可糟糕的是，它包含了一个致命的缺陷。

事实证明，java.sql.Statement.close()可能会抛出一个 SQLException 异常。它几乎不会发生，Oracle 的驱动程序只在遇到 IOException 异常尝试关闭连接（比如在数据库失效备援操作后）时才会这样做。

假设这个 JDBC 连接是在失效备援发生前创建的，用于建立连接的 IP 地址会被从一台主机转移到另一台主机，但是当前的 TCP 连接状态并不会转移到第二台数据库服务器上。当操作系统和网络驱动最终确定这个 TCP 连接已死后，所有的套接字写操作都会抛出 IOException 异常。这意味着，资源池中的每个 JDBC 连接都是随时可能发生的事故隐患。

令人吃惊的是，这个 JDBC 连接仍然可以创建数据库语句。驱动程序的连接对象只需检查自身的内部状态即可创建数据库语句。[①]如果 JDBC 连接认为它仍然处于连接状态，那么它就会创建这个语句。当做了一些网络输入输出操作时，执行这个语句就会抛出 SQLException 异常。但是，关闭这个语句也会抛出 SQLException 异常，因为驱动程序试图命令数据库服务器释放与这个语句相关的资源。

简而言之，驱动程序会创建不能被使用的语句对象。你可能认为这是一个 bug，很多航空公司的开发人员都这么认为。但是，这里得到的关键教训是，JDBC 规范允许 java.sql.Statement.close()抛出 SQLException 异常，所以你的代码也必须处理它。

在前面的出错代码里，如果关闭抛出异常的语句，连接依然不会关闭，从而导致资源泄漏。当发生 40 次这样的调用后，资源池就耗尽了，随后的所有调用都会阻塞在 connectionPool.getConnection()。这就是我在 CF 线程转储中看到的。

一个有数百架飞机、价值数百亿美元的跨国航空公司和数千名员工都被程序员的一个初级错误击倒了：一个未捕获的 SQLException 异常。

2.5 一点预防

当这样的小错误带来如此惊人的损失时，自然的反应是："以后决不能再发生这样的事了。"但是，该如何预防呢？代码审查能防止这样的错误吗？只有在审查者中有人知道 Oracle 的 JDBC 驱动程序内部机制，或者审查组对每个方法都花数小时审查时，这才可能有效。更多的测试能预防这样的错误吗？也许吧。一旦问题确定，测试组会在压力测试环境下展开测试，以证明没有同类错误了。常规的测试配置不会对这样的错误进行充分的测试，换句话说，一旦你知道错误在哪了，那就很简单，展开一项测试去发现它。

① 这可能是 Oracle 的 JDBC 驱动程序所特有的问题。我反编译出的代码只有 Oracle 的是这样的。

终究，这只是幻想，幻想每一个这样的错误都能被查出。错误还会发生，不能被完全清除，所以它们还能存活。

最糟糕的问题是，一个系统的漏洞可能会传播到其他受影响的系统。最好要问一问："我们该如何阻止一个系统的错误影响到其他的系统？"今天的每个企业内部都是一个相互联系、相互依存的系统，它们不可能，也必须不能允许错误带来链式反应。后面你会看到有一些阻止这类错误蔓延的设计模式。

第3章
稳定性概述

新软件就像是刚毕业的大学生，乐观，有活力，却要突然面对象牙塔之外的残酷现实。那些不存在于实验室而发生在现实中的事情，通常都是不好的事情。在实验室里，所有的测试都是由知道自己期望结果的人来设计的，而在现实世界中，系统面临的测试并没有预先知道的期望结果。有时候，它们将软件配置好之后，面对的却是系统的失效。

企业级软件必须学会“玩世”，“玩世”软件料到会发生一些不好的情况，对于糟糕的情况它们从来都不会大惊小怪。它们甚至都不相信自己，因此建立起内部壁垒以保护自身免受失败。它们甚至拒绝与其他系统连接太紧密，因为走得太近就会“受伤”。

上一章所讨论的航空公司核心设施项目就不够“玩世”。团队一如既往地狂热追求新技术和高级架构，热衷于谈论杠杆作用和协同作用。他们被美元冲昏了头，没有留意路旁的停车标志，结果一掉头就出了车祸。

稳定性差导致了高成本，最明显的成本就是收入的损失。在第1章提到的零售商，即便是淡季，每停机1小时也要损失10万美元。甚至只丢掉一笔交易，就可能会损失这么多。

根据经验数据，网上零售商每获得一个客户，就要花费25～50美元的成本。假设每小时有5000个独立访客，其中有10%的潜在客户会离开，那也意味着12 500～25 000美元的损失。[①]

不太明显但是同样令人痛苦的是，你因此失去了信誉。品牌的褪色并不能立即导致顾客流失，但可能会让你的假日季节运营问题报道在《商业周刊》上。为招徕顾客花费数百万美元做了形象广告，可能会在几小时内被一批坏的硬盘驱动器所抵消。

稳定性好不一定要有巨大的付出。在构建系统的架构、设计和低级别实现中，有许多决策点能充分影响系统的最终稳定性。在面临这些权衡点时，两条路都可能满足功能需求（达到QA的目标），一个将导致每年数小时的停机时间，而另一个则不会。令人惊奇的是，高度稳定的设计通常和不稳定的设计的实现成本相同。

① http://retailindustry.about.com/library/weekly/aa122599a.htm。

高度稳定的设计通常和不稳定的设计的实现成本相同。

3.1　定义稳定性

说到稳定性，我需要定义一些术语。事务是系统处理的一个抽象工作单元，它并不等同于数据库系统的事务。单个工作单元可能包含很多数据库事务。例如对于电子商务网站，一个共同的事务类型是“客户下订单”。这项事务可以跨越几个页面，通常包括一些外部功能的集成，比如信用卡验证。事务就是系统存在的原因。只处理一种事务的系统，是一个专职系统。而混合型系统是融合多个不同事务类型的系统。

我使用*系统*这个词，指的是处理用户事务的硬件、应用程序和服务三者组成的完整且相互依存的集合。一个系统可能只是一个简单的应用程序，也可能是一个庞大的多层次应用程序和服务器网络。

当要指代从头到尾处理事务的主机、应用程序、网络、电源等整个集合时，我就使用*系统*这个词。

在有短暂冲击、持续压力或者组件的故障破坏了处理流程的情况下，一个可伸缩的系统依然能够处理事务。这就是大多数人所表达的*稳定性*。它不是指你的个人服务器或应用程序持久运行，而是指用户仍然能够完成工作。

*冲击*和*压力*这两个词来自机械工程。冲击是指对系统的一次快速打击，就像用锤子敲击一样；而系统的压力是指系统长时间所受到的力。

由于听信了一次关于优惠的谣言，快闪族们蜂拥到 Xbox 360 产品的详细页面上，造成了冲击。1 分钟内突然到达上万个新的会话，系统是很难承受的。被 Slashdot 引用也会引起这种冲击。在 11 月 21 日午夜，转储 1200 万条消息到消息队列里是一种冲击。这些冲击都能够在眨眼间毁坏系统。

另一方面，由于没有足够的能力处理所有用户请求，来自信用卡处理器的慢反应是一种对系统的压力。在机械系统中，施加压力时会造成材料变形。这种形状的变化称为*应变*（劳损）。压力带来应变。同样的事情也发生在计算机系统中。信用卡处理器的压力会导致系统其他部分的应变，可能产生奇怪的效果。它可能对网络服务器带来更高的 RAM 占用量，或者对数据库或其他服务器带来过高的 I/O 占用率。

一个长寿的系统能够长时间处理事务。什么是长时间？这要看情况而定。一个比较实用的定义是指代码部署之间的时间。如果每星期都有新代码被部署，那么系统是否能保持运行两年而不重启就不重要了。另一方面，蒙大拿州西部的一个数据采集系统确实不需要每星期都手动重启一次。（除非你想住在蒙大拿州西部。）

进行寿命测试，这是找到长期存在的错误的唯一方式。

延长生命期

系统寿命受到的主要威胁是内存泄漏和数据的疯涨。这两大杀手会在生产环境中杀死系统，而且在测试时很少能被捕获。

测试会暴露问题，以便你能解决问题，正因如此，当测试人员发现错误时我总是要感激他们。有条墨菲定律，只要你不针对某种失效情况特别做测试，它就一定会发生。因此，如果你不测试，午夜崩溃、系统运行 49 小时内存耗尽等故障都会发生。如果你不测试以保证七天后没有内存泄漏，那么七天后就一定会出现内存泄漏。

麻烦的是，应用程序无法在开发环境中运行足够长的时间，以暴露长期存在的错误。通常你在开发环境中保持应用服务器运行多长时间？我敢打赌，平均寿命肯定不超过一集 TiVo 情景喜剧的长度[①]。在 QA 时，它运行的时间可能会长点，至少每天都要重启，甚至还更频繁。而且就算它运行起来，也不会处于连续负载的状态下。这些环境都不能算长期运行测试，比如在日常流量下让服务器运行一个月。

这些类型的错误通常也不会在负载测试中发现。负载测试是指运行指定的一段时间后退出。由于负载测试供应商通常每小时都要收取大量费用，所以没人会请他们每次保持负载运行一个星期。你的开发团队很可能共享着公司的网络，所以你不能都耽误公司的重要活动（比如电子邮件和上网）一连好几天。

那么，你该如何找出这些错误呢？唯一的方法是未雨绸缪，先长时间运行系统，也就是做寿命测试。如果可能的话，预留一台开发机，给它安装上 JMeter、Marathon 等负载测试工具。不要给系统很大的负载，仅仅一直保持系统有请求驱动。（而且，确保测试脚本中每天有几个小时降低请求速度来模拟午夜时那段空闲期，这将捕获连接池和防火墙超时。）

有时候，经济条件不足以建立一个完整的环境，但至少可以尝试测试重要的部分，这总好于没有测试。

如果所有办法都实施不了，那么只好把生产环境默认变成你的寿命测试环境。你肯定能找出错误，但这显然不是幸福生活的路子。

① 去掉广告以及片头和片尾，大约是 21 分钟。

3.2 故障模式

突发冲击和过度压力都会引发灾难性的故障。不管是哪种情况，系统的某些部分都会在其他部分出现故障前发生故障。在 *Inviting Disaster*[Chi01]中，James R. Chiles 指出，这些是系统的裂缝。他将处于故障边缘的复杂系统和存在微金属裂纹的钢板做了类比。在压力下，裂缝开始扩散，越来越快；最终，裂缝扩散的速度超过了音速，于是金属在爆炸声中断裂。最初的触发点、裂痕扩散的方式和损害的结果，并称为故障模式。

不管怎样，系统会有多种故障模式。拒绝接受“故障必然会发生”的事实，你就失去了控制和包容它们的力量；一旦接受了这个事实，你就有能力针对具体故障设计系统。正如汽车工程师设计保护乘客的吸撞缓冲区一样，你也可以创建控制损害和保护系统其他部分的安全故障模式。这种自我保护措施决定了系统的整体应变能力。

Chiles 称这些保护设计是裂痕终结者。就像创建吸撞缓冲区以吸收冲击、保护乘客的安全一样，你可以决定什么系统功能是必不可少的，然后在故障模式下构建系统以避免裂痕损害这些功能。如果你没有设计故障模式，那么发生危险时，结果就不可预知了，并且通常是有危险的。

3.3 裂痕扩散

我们来看看如何将这个方法应用于航空公司停机的案例。航空公司的核心设施项目没有设计它的故障模式。裂痕开始于对 `SQLException` 异常的不当处理，但它能够在许多其他地方得到遏制。我们看一些例子，涉及从底层细节到高层架构。

由于线程池配置为当没有可用资源时就阻塞请求线程，因此它最终可能会捆绑所有的请求处理线程。（这在每个应用服务器实例中都会发生。）线程池可以配置为在连接耗尽时创建更多的连接，也可以配置为在有限的时间内阻止调用者，而不是当所有连接被取出时永远阻塞。这两种方法都能避免裂痕扩散。

再往上一个层次看，这是一个对 CF 的调用导致其他主机上的调用程序崩溃的问题。因为 CF 作为 EJB 来暴露它的服务，所以它使用了 RMI。默认设置下，RMI 调用不会超时，换句话说，调用者会阻塞等待来自 CF 的 EJB 回应。每个实例的前 20 个调用者会收到异常，精确地说是一个 `SQLException` 异常，它被包裹在 `InvocationTargetException` 异常里，而后者又被包裹在 `RemoteException` 异常里。此后，调用就被阻塞了。

客户端可以针对 RMI 套接字设置超时。[①]在某个时间点，CF 也不得不决定建立一个基于 HTTP

① 例如，通过建立一个套接字工厂，对它新创建的套接字都调用 `Socket.setSoTimeout()`。

的网络服务，而非 EJB。然后，客户端就能对 HTTP 请求设置超时了。[1]也可以重新写客户端调用代码，丢弃受阻塞线程，而不是由请求处理线程进行外部集成调用。当这些都没完成时，裂痕会从 CF 扩散到所有使用 CF 的系统去。

在更大的范围上看，CF 服务器本身可能已分成多个服务组。这样可以使问题控制在一个服务组中，而不影响所有其他使用 CF 的系统。（在这种情况下，所有的服务组会以相同的方式出现故障，但是并不会总是这样。）这是又一种遏制裂痕扩散到其他组件的方法。

来看看更大的架构问题，CF 已经使用请求/应答消息队列建立了。这时，调用者也许知道，可能永远不会收到应答。作为处理协议的一部分，必须处理这种情况。更加彻底的是，调用者可以通过查找匹配搜索条件的元组条目寻找航班。而 CF 会持续不断地从数据库中把航班信息填充到元组中。架构耦合得越紧，代码错误扩散的机会也越大。反之，架构越松散，就如同减震器一样，越能消除而非扩大这种错误的影响。

所有这些方式都能使 `SQLException` 异常避免扩散到航班系统的其他部分。可悲的是，系统的设计者并没有在创建这些共享的服务时考虑“裂缝”存在的可能性。

3.4 故障链

在每个系统故障下，都潜藏着一条事件链。一件小事导致了另一件事，而后者也导致了另外的事。事后再看整个故障链，似乎故障是不可避免的。如果你尝试估计事件链确切发生的概率，它又似乎是非常不可能的。只有当你考虑到每个独立事件的概率时，这才看起来是不可能的。硬币没有记忆，每次抛硬币出现正面或者反面的概率都是相同的，与之前抛硬币无关。导致故障的事件组合并不独立。一个点或者一个层上的故障实际上增加了其他故障的可能性。如果数据库变慢，那么应用程序更可能出现内存不足。因为层与层之间是耦合的，所以事件并不独立。

在故障链的每一步，裂痕都可能加速、变慢或停止。高层次的复杂性为裂痕的扩散提供了更多的方向。

紧耦合加速了裂痕。例如，EJB 调用的紧耦合带来了 CF 中的资源枯竭问题，这也给调用者带来更大的问题。这些系统将请求处理线程与外部调用耦合起来，使远程的问题演变成停机。

为每种可能的故障做好准备的一种方法是，检查每个外部调用、每次 I/O、每次资源的使用以及每个预期的结果，并问一问：“还有哪些方式可能出问题？”考虑下面各种可能存在的冲击和压力：

- 如果我不启动连接？

① 除非它能使用 `java.net.URL` 和 `java.net.URLConnection`。在 Java 5 之前，它还不能对标准 Java 库的 HTTP 调用设置超时。

- 如果连接需要 10 分钟?
- 如果已经连接，那它如何断开连接?
- 如果已经连接，而我却不能得到另一端的任何回应?
- 如果需要 2 分钟来回应我的查询请求?
- 如果同时有 10 000 个请求?
- 如果网络陷入蠕虫的攻击而瘫痪，当我要将关于 `SQLException` 异常发生的错误消息写入日志时，我的磁盘已满?

我已经累了，可这只是可能出错的开始。因此，蛮力的方法对于除了生命攸关的系统和火星探测器之外的系统来说都是不切实际的做法。但如果你在这十年内需要交付一个这样的系统，该怎么办呢? 你需要找到一些模式来创建减震器，释放那些压力。

3.5 模式与反模式

我已经处理了数百个生产故障，每一个都是独一无二的。(它们大多是独特的，因为我尽量不让相同的故障发生两次!) 我没见过相同方式发生的完全相同的故障链：相同的触发器，相同的断裂，相同的扩散。然而，随着时间的推移，故障的模式确实会出现。沿着某个轴表现出来的脆弱性，一个问题会放大另一个系统行为的趋势。这些都是稳定性的反模式。第 4 章会讲述这些故障模式。

如果存在故障的系统模式，你可以想象，能使用一些共同的解决方案。对，第 5 章描述了一些抵御反模式的设计和架构模式。这些模式并不能防止系统的裂缝，没有什么能做到这个，总会有一些条件的集合能触发裂缝。这些模式阻止裂缝的传播，它们帮助遏制裂缝的破坏并保护部分功能，而不是让系统崩溃。

毫不奇怪的是，这些模式和反模式彼此相互影响。反模式有加强彼此的倾向。就像对付电影中各种怪物的大蒜、银器和火一样，[①]每个模式都缓解了某些问题。

图 3-1 展示了这些交互中最重要的部分。现在你会看到故障的共同来源：反模式。

① 它们分别能克吸血鬼、狼人和弗兰肯斯坦医生创造的怪物。

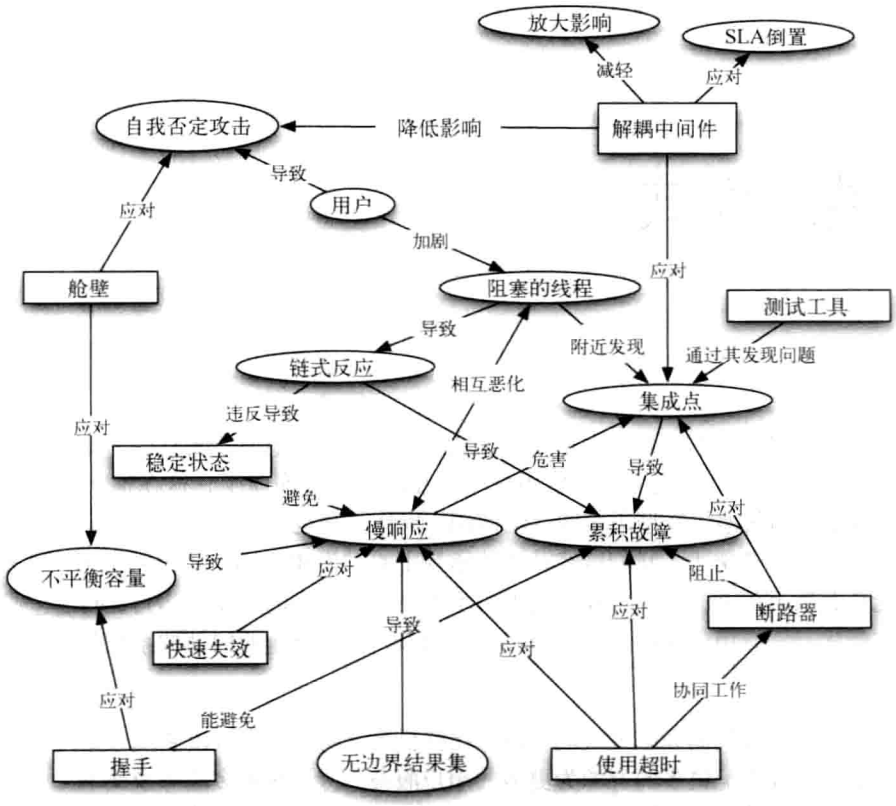


图 3-1 模式与反模式的交互

第4章
稳定性反模式

曾几何时，应用程序崩溃是最常见的系统错误，操作系统崩溃紧随其后。如果我说这种状况多年来都没改变，显然是在说谎了。应用程序现在已经很少崩溃了，这在很大程度上要归功于Java、PHP、Ruby和其他解释型语言的广泛使用。操作系统也已经变得更加稳定可靠了，这应归功于广大程序员的辛苦努力。我们曾经认为能承受上百并发用户的就是大系统了，可如今，很多系统已经能负载数万的并发用户。我们追求的不再是程序数小时的正常运行，而是它们能保持数月时间的连续运行。应用程序的规模也呈爆炸性增长，首先是企业内部的系统集成，其次是跨企业的整合。

当然，这也意味着我们面临更大的挑战。在整合系统时，应当做出紧耦合系统，而不是异常系统。大型系统通过支配更多的资源为更多的用户服务；但是在很多故障模式下，大型系统比小型系统崩溃的速度更快。这些系统的规模和复杂性促使我们研究 *Inviting Disaster*[Chi01]所称的技术前沿是什么。高交互复杂性和紧耦合这两大相生相伴的隐患，共同将系统的裂缝扩散迅速转变成完全崩溃。

当系统存在足够多的移动组件且被隐藏时，高交互复杂性就出现了，大多数操作者关于内部依赖关系的心智模式是不完整或者完全错误的。在《设计心理学》[Nor88]中，Don Norman描述了用户心智模式与实现模式的断裂：当实现过程不可见而外观表现又不明显时，这种断裂就可能出现。他用冰箱控制冷藏室和冷冻室温度的两个刻度盘做例子。心智模式下，调整刻度盘能让牛奶冷冻，肉类解冻，因为实际机制是控制送入每部分的冷气。对一个高交互复杂度的系统来说，操作者的直觉行动可能是无效甚至有害的操作。根据操作者自身心智模式对系统功能的认识，即便完全出于好意，他也可能触发完全预料之外的连锁反应，这种连锁反应能够直接导致问题膨胀，将一个小问题变成大事故。“三里岛核事故”[①]应该部分归咎于冷却检测与控制系统的隐藏连锁。这些隐藏的连锁通常在事后分析中很明显，而实际上却很难预料。

紧耦合使系统某部分裂缝跨层或跨系统边界向外扩散。现实生活中的例子可以说明这个问

① *Inviting Disaster*[Chi01]，37～63页。

题：四个螺钉穿过一块金属板支撑起了一个展台。展台、螺母和螺栓、金属板和顶板明显是紧耦合的。（事实上，这正是螺栓的特点！）单个螺栓出现故障，会给其他螺栓、顶板和展台增加很大的压力。增加的压力极可能让系统其他组件出现故障，而且可能就是展台本身出问题。在你的系统里，紧耦合可能出现在应用程序代码内、系统间调用，或是多个用户共享同一资源的任何地方。

在这一章，我们将看看我观察到的 11 个稳定性的反模式，这些都是常见的因素，它们是导致多个系统发生故障的根源。其中有些因素就像是一个人一边用锤子敲自己的脑袋，一边和医生说："医生，每当我这样做就很疼。"医生回答道："不要这样！"每个反模式将会产生、促进或增加系统的裂缝。我们需要避免这些不良行为。

然而，在一般情况下，我们最需要记住的是：事情总会有纰漏。不要假设你可以消除一切故障的根源，因为无论是与生俱来还是后天形成，都会出现更大的故障破坏你的系统。请做最坏的打算，因为裂缝随时都会发生。

反模式将会产生、促进或增加系统的裂缝。

4.1 集成点



大约从 1996 年开始，我就没再见过"纯网站"项目。如果你的项目与我的相似，那么它们很可能是一个碰巧基于 HTML 前端的企业集成项目。确实，在真正需要创建动态网站之前，许多公司并不会开始企业集成，他们只会口头上说说。那些项目是最终促使企业对此前协作不佳的系统进行集成的动力。看看这些项目的系统环境图，你会发现网站在图的中心，相连的线向各个方向延伸。数据来自于仓库、标价、内容管理、CRM、ERP、MRP、SAP、WAP、BAP、BPO、R2D2 和 C3P0。提取的数据广泛应用于 CRM、实现、预订、授权、欺诈核查、地址标准化、调度、运输等。最终，为商务人士生成商业统计报表，为技术人员提供技术统计表，为管理者构建管理统计报表。

集成点是系统的头号杀手。任何一个输入都存在稳定性风险。每个套接字、进程、管道或远程过程调用，都能够而且随时可能会出现故障。甚至数据库调用也可能中止，只是有时候很明显，有时候不容易察觉而已。进入系统的每个输入都可能导致系统中止、崩溃，在最坏的情况下还可能产生其他冲击。我们接下来就看看使集成点表现不良的一些具体方式，以及你该如何应对。

基于套接字的协议

许多高层集成协议都在套接字上运行。实际上，除了已命名管道和共享内存的 IPC 之外，几乎所有协议都基于套接字。高层协议带有其特有的故障模式，但它们也都易受套接层故障的影响。

在远程系统拒绝连接时，最简单的故障模式就发生了。调用系统必须处理连接失败。通常这不是什么大问题，因为不管是 C、Java 还是 Ruby，都有明确的方式表明连接失败：在 C 中返回

值是–1，在 Java/C#/Ruby 中显示为异常。因为 API 会明确表示连接不是一直都有效的，所以程序员能处理连接无效的情况。

有多少输入

我曾帮助一个大型零售商启动了一个重新构建平台/架构的项目。我们为所有产品建立防火墙规则，以便打开防火墙，为产品系统进行授权通道的连接。我们通过了常规的检查：网络服务器连接到应用程序服务器，应用程序服务器连接到数据库服务器，集群管理器连接到集群节点，等等。

当需要为生产环境添加输入输出规则时，我们被介绍给负责企业集成的项目经理。没错，这个网站重构项目有负责集成的项目经理。于是我们得到了第二条线索，知道这不是个容易的任务。（第一条是，没人能告诉我们输入输出是什么。）项目经理准确理解了我们的需求，他运行集成数据库，为我们输出了一份所有连接细节的定制报告。

一方面，他有一个如此完备的数据库来跟踪各种输入输出（同步/异步，成批或分段处理，源系统，频率，容量，交叉引用的数字，业务权益方等），让我印象深刻。但另一方面，我对他居然需要数据库来跟踪这些东西感到很失望。

当网站启动时，它被稳定性问题困扰也就不足为奇了。就像家里的新生婴儿让你无法安睡一样，每到夜里 3 点我就被系统崩溃或危机叫醒。我们需要不断记录程序崩溃的地方，并反馈给维护团队进行修正。我没有详细记录，但可以肯定每个同步的集成点都至少引发过一次运行中断。

需要注意的是：发现连接不上可能需要花很长时间。我们来快速了解一下 TCP/IP 网络的细节知识。

每个架构图都有框和箭头，如图 4-1 所示。同我们处理的其他问题一样，这个箭头代表一个抽象的网络连接。事实上，这意味着它代表的是一个抽象物体的抽象概念。网络“连接”是一种逻辑结构，而连接本身就是一种抽象。在网络里，你见到的所有东西都是数据包[①]。这是 TCP/IP 协议族的网际协议（IP）部分。传输控制协议（TCP）是一种关于如何基于离散数据包使某些东西看上去像是连续连接的协议。图 4-2 显示的是 TCP 定义的打开一个连接的“三次握手”协议。连接开始时，调用者（通常是客户端，即便对其他程序来说是服务器端本身）给远程服务器的端口发送一个 SYN 包。如果没有程序监听这个端口，远程服务器会立即发回一个 TCP“reset”包，表明服务器没有响应。调用程序会因此得到异常或者错误返回值。这一切发生得非常迅速，如果两台机器连接在同一交换机上，这一流程在 10 毫秒以内就可以完成。

① 当然，“包”也是一个抽象概念。在线路上，它就是电子。在电子与 TCP 连接之间，有很多抽象层。好在我们可以在任何给定时间适当选择任何有用的抽象层。

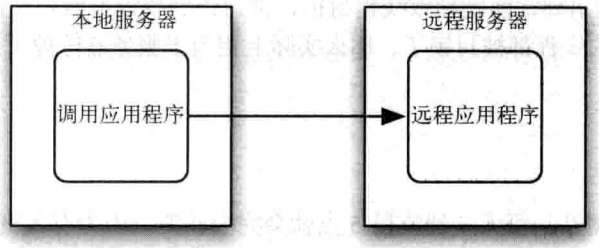


图 4-1 最简单的拓扑结构：直接连接

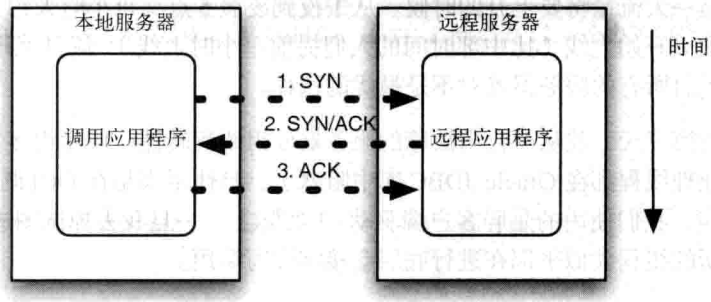


图 4-2 三次握手

如果有程序在监听目的端口，那么远程服务器会发送回 SYN/ACK 包，指明它愿意接受连接。调用方获得 SYN/ACK 包，再发回 ACK 包。这三个数据包就建立了“连接”，然后应用程序就可以传输和接收数据了[①]。

假定远程应用程序正在监听端口，而且不断地接收连接请求，直到其不能响应到达的请求为止。端口本身有一个监听队列，里面定义了网络栈支持多少未决的连接（发送了 SYN 包，但是没有 SYN/ACK 包返回）。一旦监听队列满了，进一步的连接请求就会马上被拒绝。监听队列是最拥挤的地方。当套接字处于半建立状态，调用 `open()`的线程会在操作系统内核被封锁，直到远程应用程序接受连接或连接超时。连接超时在不同操作系统中不同，但通常以分钟计！调用应用程序的线程可能被封锁长达 10 分钟，来等待远程服务器的响应。

当调用者连接并发送请求，但服务器花很长时间读取请求并发送响应时，就会发生以上类似的情况。`read()`调用会封锁直到服务器发送响应。在 Java 中，默认设置永久封锁。你不得不调用 `Socket.setSoTimeout()`来切断封锁的调用，那时，你需要准备接受 `IOException` 异常。

网络故障可能用快慢两种方式袭击你。快速的网络故障可能在调用代码中立即产生异常。“连接被拒绝”就是一个非常快的网络故障，只需几毫秒就能返回给调用者。较慢的网络故障，比如

① TCP 也定义了“同时开放”握手，这是指两台机器在收到 SYN/ACK 包之前都发送 SYN 包给对方。在基于客户端/服务器交互的系统上，这还是相对少见的。

丢弃的 ACK 包，在抛出异常前就导致线程封锁，被封锁的线程不能处理任何事务，因此整体效率降低了。如果所有的线程都被封锁了，那么实际上相当于服务器停掉了。很显然，慢响应比没有响应严重得多。

凌晨 5 点的问题

我曾开发的一个网站，每天一到凌晨 5 点就会完全崩溃。因为有大概 30 个不同的应用程序实例运行，因此一定有什么原因导致所有的应用程序实例在 5 分钟之内都挂起了（URL ping 工具的解析）。重启应用程序服务器总能清除故障，因此应该是某个瞬时效应让服务器挂掉了。不幸的是，那时正好是一天流量将要上升的时候。从午夜到凌晨 5 点，每小时大约只有 100 个事务，但是一旦东海岸的人开始上线（比中部时间的人们提前一小时上线），流量就迅速升高。在人们正要上网的时候重启所有的服务器绝对不是最优的选择。

在事故发生的第三天，我从一台出故障的服务器取得线程转储。应用程序实例运行起来了，但是所有的请求处理线程都在 Oracle JDBC 库中阻塞了，具体说来是在 OCI 调用内。（由于其优越的故障切换功能，我们使用的是胖客户端驱动。）实际上，一旦我去掉试图进入同步方法而被阻塞的线程，活跃的线程就似乎都在进行底层套接字读写调用。

接下来是使用 `tcpdump` 和 `ethereal`[①]，奇怪的是没发现什么。一些包从应用程序服务器发送到数据库服务器，但没有应答。从数据库服务器到应用程序服务器也什么都没有。然而，监测表明数据库是活跃而且健康的。目前也没有阻塞锁，运行队列里什么都没有，I/O 速率也很小。

抓包

抽象提供了简洁的表达。比起讨论建立连接、数据包分帧、确认和接收窗口等繁琐的细节，讨论 URL 提取文件要快得多。然而，对于任何抽象的东西，总有一天你不得不忍着泪水剥开洋葱，来一探究竟——通常是出问题了你才会这样做。不管是故障诊断还是性能调优，抓包工具都是搞清楚网络问题的唯一方法。

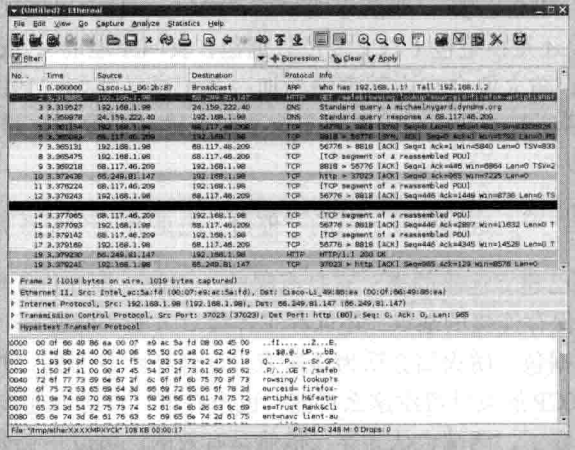
`tcpdump` 是一个常见的 UNIX 抓包工具。“混杂”模式下运行会命令网络接口卡（NIC）接收所有跨链路的包，甚至是发送给其他计算机的数据包。（在数据中心，NIC 几乎总是连接到一个分配给虚拟 LAN（VLAN）的交换机端口。这种情况下，交换机能保证 NIC 接收的包仅限制在那个 VLAN 的地址范围内。这是一项重要的安全措施，它能防止一些坏家伙模仿我们正在进行的操作：嗅探链路寻找“有用”信息。）Wireshark[①] 是一个网络嗅探与协议分析的综合工具，它能像 `tcpdump` 一样嗅探数据包，但 Wireshark 功能更多，它还能解包。Wireshark 存在大量安全缺陷，有些无关紧要，有些很严重。

① Ethereal 已经更名为 Wireshark。

曾经，攻击者使用专门制作的数据包通过网络（例如利用一台被攻破的台式机上的恶意软件）触发了缓冲区溢出，并执行他们设计的任意代码。由于 Wireshark 必须以 root 身份使 NIC 在混杂模式下运行，任何抓包工具都是如此，因而攻击者能够利用这一特点获得访问管理员机器的 root 权限。

除了这些安全问题，Wireshark 还是一个大而笨重的 GUI 程序。在 UNIX 上，它需要一堆的 X 库（甚至一些无头系统[②]上并没有安装的库）。在任何主机上，它都要消耗大量的内存和 CPU 周期来解析和显示数据包。这是一个负担，不该出现在生产服务器上。由于这些原因，最好使用 `tcpdump` 进行非交互式的抓包，然后将捕获的文件转移到非生产环境进行分析。

下面的截图是来自于 Ethereal 分析我的家庭网络上捕获的数据包。第一个包显示了地址路由协议（ARP）请求，这碰巧是从我的无线网桥到调制解调器上的问题。下一个包就奇怪了：一个对 Google 的 HTTP 查询，请求一个带一些查询参数的名为/safebrowsing/lookup 的 URL。后面两个包显示了对主机名“michaelnygard.dyndns.org”的 DNS 查询和响应。第五、六、七数据包是进行的 TCP 三次握手连接。我们可以在浏览器和服务器之间跟踪整个会话。注意包跟踪下面的面板，它显示了第二个包中封装的层，这是 TCP/IP 协议栈针对 HTTP 请求建立的。最外层的帧是一个以太网包，它包括一个 IP 包，IP 包又包括了一个 TCP 包。最后，TCP 包包含了一个 HTTP 请求。整个包的精确字节显示在第三个面板中。



我强烈建议你买一本《TCP/IP 指南》[Koz05] 或《TCP/IP 详解》[Ste93]，以便随时参考。

① http://www.wireshark.org

② 无头系统（headless system）是指独立运行的计算机，不需要人类的直接输入，此类计算机不需要终端。——编者注

这时不得不重启应用程序服务器。我们优先恢复服务，可以的话，我们会收集数据，但是不会以违反 SLA 为代价。等事故再次发生，我们才会做进一步的调查，大家都明白事故肯定会再次发生。

服务水平协议：提供一个可衡量、量化的服务的合同义务，违反 SLA 会受到经济处罚。

果然，第二天早上事故依旧上演。应用程序服务器锁得紧紧的，就像鼓，一堆线程挂在 JDBC 驱动程序上。这次，我能看到数据库网络上的流量接近于零。什么也没有。在防火墙一端完全缺乏流量就如同夏洛克·福尔摩斯的狗在夜里没有叫一样——活动的消失往往就是最大的线索。我有了一个假设，立刻反编译服务器资源池类的代码，证明了我的假设是合理的。

我前面提到，套接字连接是抽象的，它们只是作为端点处计算机内存中的一个对象存在。TCP 连接一旦建立，可能持续数天，即使连一个包都不发送[①]。只要两台计算机在内存中都保持套接字状态，“连接”就仍有效。路由可变，物理链路可被切断和重连。但这没关系，只要两端的计算机认为“连接”状态还在，那么它就存在。

曾经一切都工作得完美无缺。可现在，一些偏执的小壁垒已经瓦解了整个网络的思想体系和实现。没错，我说的就是防火墙。

防火墙就是一个特殊的路由器，它将数据包从一个物理端口路由到另一个物理端口。每个防火墙里都有一套访问控制列表，定义了它允许连接的规则，比如，“允许从 192.0.2.0/24 到 192.168.1.199 的端口 80 建立连接”。当防火墙收到 SYN 包，它会根据规则检查，来判断是允许包（路由到目的网络），还是拒绝包（TCP 复位数据包，发回原地址），或是忽略包（丢弃包而不回应）。如果连接允许，防火墙会在内部表中添加一条记录，比如“192.0.2.98:32770 被连接到 192.168.1.199:80”。随后所有的数据包，不管是来自哪儿的，只要与连接的两端匹配，就能够通过防火墙网络进行连接。

目前为止一切都好，可是这和凌晨 5 点的“催命”电话有什么关系呢？

关键在于防火墙内已建立连接的表。它是有限的，因此，不允许无限期的连接，尽管 TCP 本身允许。防火墙不仅保存连接的端点，还保存了连接双方的“最后一个包”的抵达时间，如果一个连接长时间没有数据包，防火墙会认为连接双方已经死亡或消失，并将该连接从表中删除，如图 4-3 所示。但是，TCP 的设计可没这么智能，也没有第三方通知连接的两端，连接已经断开了。即便没有数据包传输了，两端也认为连接依然是有效的，可以一直如此。

此后，由于套接字的半开状态，从任何一端进行套接字读写操作都不会导致 TCP 重置或者报错。相反，TCP/IP 协议栈会发送数据包，等待 ACK 应答，如果没有响应就重发。忠实的协议栈一遍遍地尝试重新连接，而防火墙只是把数据包都丢弃了，而没有发送类似“ICMP 目标不可

① 假设你在内核中设置了合适的反常超时。

达”这样的消息。(这可能会让攻击者通过假冒的源地址嗅探到活跃的连接。)我的 Linux 系统运行的是 2.6 系列内核，将 tcp_retries2 设置为默认值 15，这使得 TCP/IP 协议栈在通知套接字库连接断开之前有 20 分钟的超时。我们使用的 HP-UX 服务器是 30 分钟超时，也就是说应用程序一行套接字写操作可能阻塞 30 分钟！套接字读操作在这种情况下更遭，可能永远阻塞。

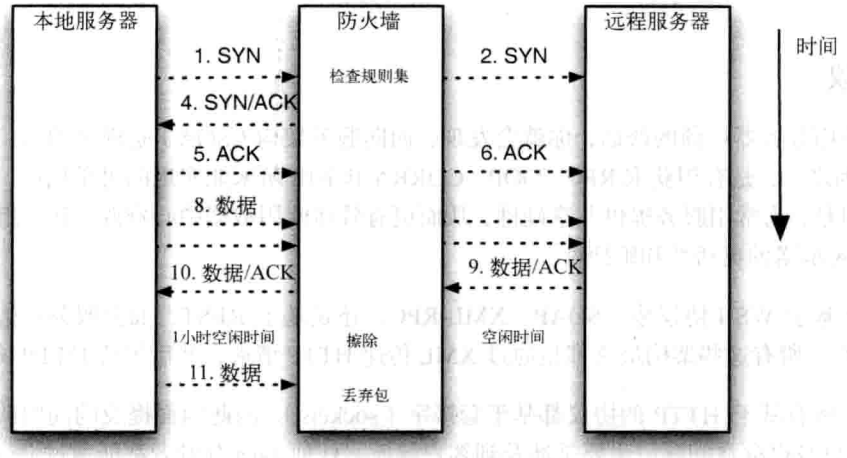


图 4-3 被防火墙丢弃的空闲连接

当我反编译资源池类时，我意识到它采用了后进先出的策略。网站深夜流量小，单个数据库连接会被从池中签出，使用，然后签入。下一个请求可以获取同一个连接，在流量开始上升之前，其余的 39 个连接都会闲置。防火墙的空闲连接超时被设置为 1 小时，而 39 个连接空闲的时间远远超过了 1 个小时。

一旦流量上升，每个应用程序服务器的其余 39 个连接就会立即锁起。即使这一个连接仍然在使用，它迟早会被一个线程签出，而该线程最终会被来自其他资源池的某个连接阻塞，唯一好用的连接也被阻塞线程占用了，最终整个网站也就挂了。

一旦我们理解了故障链的每个环节，接下来就该找到解决方案了。签出连接前，资源池能够测试 JDBC 连接的有效性。执行类似“SELECT SYSDATE FROM DUAL”这样的 SQL 查询即可验证有效性，但这会将请求处理的线程挂起。我们本可以在池中记录 JDBC 连接的空闲时间并丢弃空闲超过 1 小时的连接，但不幸的是，这还需要发送一个数据包，告诉数据库服务器会话正在被断开。很明显，以上两种解决方案都会导致服务器宕机。

我们开始考虑一些比较复杂的解决方案，比如创建一个 reaper 线程，寻找那些驻留时间过长的连接，并赶在超时之前关闭它们。Oracle 有一个叫*死连接检测*的功能，它能发现客户端崩溃的时间。启用这个功能，数据库服务器每隔一段时间会给客户端发送一个 ping 包，如果客户端响应了，则数据库知道它还是活跃的，否则重复发送多次还未收到响应，数据库就认为客户端已经

崩溃，然后清空该连接占用的所有资源。

我们并没有那么担心客户端崩溃，因为 ping 包本身足以重置防火墙的“最后一个包”的抵达时间，用来保持连接有效。死连接检测能保持连接活跃，我终于可以睡个踏实觉了。

接下来，我们会看看基于 HTTP 协议的问题，包括网络服务。

HTTP协议

听听应用服务器厂商的谈话，你就会发现，面向服务架构（SOA）是现在的热门话题。追求 SOA 的原因之一，是有望获取 RPC、OOP、CORBA 和 EJB 所未能实现的可重用性。另一个常被提及的原因是，为常用服务提供共享硬件，从而更有效地使用数据中心资源。其他组织则可能是看中了 SOA 承诺的灵活性和敏捷性。

不管是基于 WS-I 协议族、SOAP、XML-RPC，还是基于 REST，面向服务架构的共同特点都是 HTTP[①]。所有这些架构最终都是通过 XML 传递 HTTP 请求，然后等待 HTTP 响应。

当然，所有基于 HTTP 的协议都基于套接字（sockets），因此前面提及的问题缺陷都可能存在。HTTP 自身也存在问题，主要是涉及到客户端库。任何 Java 开发者都能通过 java.net.URL 和 java.net.URLConnection 类来获得内置的 HTTP 客户端。

```
Line 1   URL url = new URL("http://www.google.com/search?q=foo" );
     -   URLConnection conn = url.openConnection();
     -   HttpURLConnection httpConnection = (HttpURLConnection)conn;
     -   httpConnection.setRequestProperty("User-Agent" ,
     5   "Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.8.0.1) " +
     -   "Gecko/20060111 Firefox/1.5.0.1" );
     -   InputStream response = httpConnection.getInputStream();
```

Java 的高度通用 URL 类试图隐藏 HTTP、HTTPS、FTP 和其他协议的区别。在第 1 行，我们创建了一个到 Google 的查询 URL。第 2 行是打开连接，但实际上并没有发送请求，它只是一种工厂方法模式，用以创建真正发送 URL 连接请求的具体子类。我们得将返回的 URLConnection 对象向下转换为具体的子类对象，然后调用第 4 行的 `setRequestProperty()`方法[②]。最后在第 7 行，`HttpURLConnection` 打开一个套接字到远程主机，发送 HTTP 请求，等待和解析 HTTP 响应，再返回 `InputStream` 对象。

第 7 行实际上发生了很多事，这是一个大阻塞调用，没有机会设置任何参数。举例而言，如果你想通过调用 `Socket.setSoTimeout()`方法来设置套接字超时，那就得创建 `Socket`

① 从技术上讲，SOAP 和 WS-I 协议都允许其他消息的传输，但实际上，只有 TIBCO 和 IBM MQ 的开发者在使用它们。更多的人应该是在使用异步消息传输。

② 我们不得不隐瞒我们的用户代理，否则 Google 会返回 403 “Forbidden” 响应！

ImplFactory 对象来影响所有的套接字，而不仅仅是针对这个 HTTP 交互的套接字。而且请注意，getInputStream()调用没有超时。远程系统可以在未来十年里每秒只返回一个字节，那么你的线程将仍然阻塞在这个调用上。

一个老练的系统绝不会容忍这种无保护的调用。幸运的是，其他可用的 HTTP 客户端都允许更多的控制。比如，Apache Jakarta Common 的 HttpClient 包提供了连接和读超时的精细控制，还能控制请求首部、响应首部及 cookie 策略。

厂商API库

有人认为企业软件厂商肯定针对 bug 强化了其软件，因为他们就是卖软件的，并且它们的软件会安装在很多客户端上。这种想法很美好，且对于他们出售的服务器软件来说可能是这样，但是客户端库就很少如此了。通常，软件厂商提供客户端 API 库，这些库问题很多，而且通常隐藏着很多稳定性风险。这些库只是普通开发者编写的代码。他们在质量、样式和安全性上都不同，就跟你看到的其他代码的随机抽样一样。

最糟糕的是，你几乎无法控制这些库。最多就只能对代码进行反编译，发现问题，然后报告错误。如果你有足够的影响力，能向厂商施加压力，而且假设你使用的是其最新版软件，那么就可能让他们修复客户端库上的错误。过去，我会修复他们的错误，然后重新编译自己的版本，暂为使用，直到厂商出补丁版本，我还因此出了名。

厂商 API 库稳定性的头号杀手就是阻塞。不管是内部的资源池、套接字读取调用、HTTP 连接，抑或普通的 Java 序列化，厂商 API 库都充斥着各种不安全的代码。

下面就是一个典型例子。一旦出现多个线程需要同步多个资源的情况，就有死锁的风险。线程 1 持有锁 A 而需要锁 B，同时线程 2 持有锁 B 而需要锁 A。为了避免这类死锁，应确保始终以相同的顺序获得锁，并且以逆序释放它们。当然，只有在你知道该线程需要这两个锁并且能够控制获得锁的顺序时，这种方法才适用。让我们来看一个 Java 中的例子，这一例证可能来自于某种面向消息的中间件库:

code/stability_anti_patterns/UserCallback.java

```
public interface UserCallback {
    public void messageReceived(Message msg);
}
```

code/stability_anti_patterns/Connection.java

```
public interface Connection {
    public void registerCallback(UserCallback callback);

    public void send(Message msg);
}
```

我敢肯定这看起来很眼熟。安全吗？不知道。如果你不知道是什么线程在调用 message

Received()，那就无法确定这个线程会持有什么监视程序。栈上可能已经有十多个同步方法了，成了死锁的雷区。

即使UserCallback接口不声明messageReceived()为同步（你不可能声明接口方法同步），实现也可能令它同步。依据客户端库中的线程模型和你回调函数花掉的时间，同步这个回调函数可能会在客户端库中阻塞线程。就像堵塞排水一样，那些被阻塞的线程可能引发调用 send()函数的线程阻塞。通常，如果所有处理请求的线程都被阻塞了，那么你的应用程序可能也会宕机。

防范集成点问题

不集成任何其他东西的独立系统是罕见的，而且几乎没用。如何能使集成点更安全？防范集成点故障最有效的方法是断路器和解耦中间件。

利用断路器和解耦中间件防范集成点故障。

测试也有帮助。成熟的软件应能处理形式和功能的冲突，比如形式错误的消息头，或突然断开的连接。为了确保软件足够老练，应该为每个集成测试做一个测试工具，即提供可控行为的模拟器。设置测试工具，让其挡住千篇一律的响应，这样做有助于功能测试，也能让测试隔离在目标系统之内。最后，每个测试工具也应该能够让你模拟各种系统和网络故障。

这个测试工具能立即帮助你进行功能测试。若要进行稳定性测试，则需要在系统负荷相当大的情况下，翻转工具所有的开关。这些负荷可能来自运行 JMeter 或 Marathon 的工作站，但它肯定需要更多的测试器点击其台式机。

请记住

- 当心这个无可避免的恶魔

每个集成点最终会在某些地方出现故障，你需要为故障做好准备。

- 为多种形式的故障做准备

集成点故障有多种形式，比如各种网络错误和语义错误。你所获得的障碍不是按照既定协议产生的友好错误应答，而是一些协议冲突、慢响应，或直接中止。

- 知道何时开放抽象

调试集成点故障通常需要剥离一层抽象。故障在应用层很难调试，因为它们大多与高层协议冲突。数据包嗅探器和其他网络诊断工具能帮上你的忙。

- 故障迅速传播

如果你的代码没有足够的防御能力，那么远程系统的故障就会通过连锁故障，很快变成你的故障。

- 应用模式以避免集成点故障

断路器、超时机制、解耦中间件和握手机制这些防御性程序设计，会帮助你避免集成点危险。

4.2 连锁反应



在 8.1 节中，我会谈到两种主要的容量扩展方式：水平扩展和垂直扩展。水平扩展是指通过添加服务器来扩大容量，谷歌和亚马逊就是这样做的。网站农场就是水平扩展的例子——每台新添服务器的容量几乎和老服务器相同。还有种做法是垂直扩展，即构建更大的服务器，用 4 路、8 路、甚或 32 路的服务器替换 x86 披萨盒，这是 Oracle 喜闻乐见的做法。这两种扩展方式各有其适用条件。

如果系统进行水平扩展，那么就需要有负载平衡的服务器农场或集群，里面的每台服务器都运行相同的应用程序。多台机器由于冗余性而具有容错能力。单台机器或进程完全崩溃时，其他服务器依然能继续处理事务。

任何设备、节点或电缆，一旦故障就会导致更大系统的完全故障，这叫作单点故障（Single Point of Failure，SPOF）。例如，只有一个电源供给的服务器和无冗余的网络交换机发生的的故障都是单点故障。

不过，即便水平集群不容易遭受单点故障，除了受到自我否定攻击时（参看 4.6 节“自我否定攻击”），它们也会表现出与负载相关联的故障模式。负载平衡集群中的一个节点发生故障，其他的节点就不得不分担它的工作。例如，图 4-4 中的 8 服务器网站农场，每个节点处理 12.5%的负载。

而当一台服务器退出工作，如图 4-5 中的分布，其余的 7 台服务器必须处理总负载的 14.3%。虽然每台服务器只多处理了 1.8%的总负载，但单个服务器的负载增加了 15%。如果是一个只有 2 个节点的服务器集群，其中一个服务器出现故障，那么剩下那台服务器的工作量就会加倍。

如果第一台服务器是因为负载相关的条件而崩溃，比如内存泄漏或者间歇竞争条件，那么剩余的节点会更可能出现故障。每多一台服务器出故障，剩下的服务器便会承担更多负载，因此也更可能出现故障。

当程序中存在缺陷，就会发生连锁反应，通常是资源泄漏或与负载相关联的服务器崩溃。我们已经假设服务器处于同构层中，也就是缺陷存在于每台服务器中，因此唯一能消除连锁反应的方法就是修复潜在缺陷。就像舱壁模式一样，将单个层分成多个池，从而能将单个连锁反应分成

两个发生率不同的连锁反应，这样有时能管用。

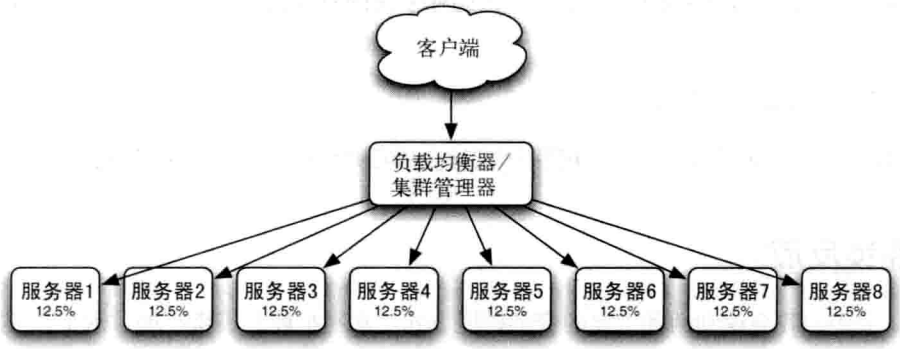


图 4-4　8 路水平服务器农场

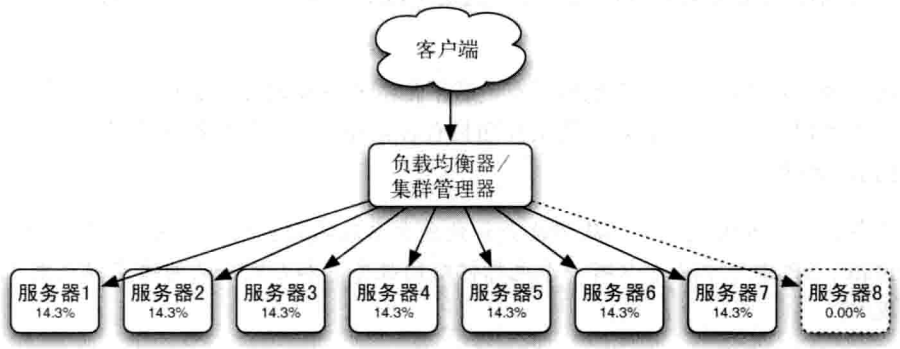


图 4-5　原来的 8 路服务器集群

连锁反应可能对系统的其余部分产生什么影响？至少，一层中的连锁反应故障很容易导致调用层的连锁故障。

连锁反应有时由阻塞线程造成。程序中所有的请求处理线程都被阻塞，且应用服务器停止响应，这时就会发生这种情况。传入的请求会分发到同层其他服务器的程序中去，这就增加了故障的机会。



请记住

- 一台服务器宕机会威胁其余的服务器

因为一台服务器宕机，其他服务器得分担它的负载，从而导致连锁反应发生。增加的负载使它们更容易崩溃。连锁反应会迅速拖垮整层的服务器，依赖于这一层的其他层必须有自我保护机

制，否则它们就会出现连锁故障。

- 寻找资源泄漏

大多数时候，连锁反应发生于程序出现内存泄漏时。一台服务器内存耗尽而崩溃，其他的服务器就会承担起崩溃服务器的工作。增加的流量意味着它们的内存泄漏得更快。

- 寻找模糊时间漏洞

模糊的竞争条件也可能被流量触发。同样，如果一台服务器出现死锁，其他服务器增加的负载也会使它们更容易死锁。

- 捍卫舱壁

通过舱壁来划分服务器，能防止连锁反应毁掉整个服务，尽管它们不会帮助崩溃分区中的调用程序。请在调用端使用断路器。

搜索

下面是我处理一个零售商的主要在线产品时发生的事情。它有一个庞大的目录，100 个不同类别里共 50 万个库存单位（SKU）。对于这个站点，搜索不仅有用，而且必须。为了应对假日期间的所有客户，零售商运行了十多台搜索引擎，它们由硬件负载均衡器控制。应用程序服务器被配置为连接到虚拟 IP 地址，而不是具体的搜索引擎。然后，负载均衡器将应用服务器的查询请求分发到搜索引擎。它也会对服务器进行正常检查，确定处于活跃和响应状态的搜索引擎，以便确保只向活着的搜索引擎发送查询请求。

这些正常检查被证明是有用的。搜索引擎中有些导致内存泄漏的错误，在正常流量（非假期）下，搜索引擎会在中午的时候崩溃。因为每个引擎在上午会承受相同的负载，它们会在同一时刻崩溃。每当一个引擎崩溃，负载均衡器就会将查询请求分摊到其他服务器上，从而加速了它们的内存泄漏。在查看它们“上次响应”的时间戳图表时，我清楚地看到了崩溃的加速趋势。第一台和第二台搜索引擎崩溃的间隙可能会有 5 到 6 分钟，而第二台和第三台的崩溃间隙只有 3 到 4 分钟。最后两台的崩溃间隙甚至以秒来计。

这套系统也遭受了连锁故障和阻塞线程故障。搜索服务器全部故障后会导致整个前端也完全死锁。我们不得不在每天的高峰时间重启服务器：上午 11 点，下午 4 点，晚上 9 点。经过数月的交涉，我们从厂商那里获得一个有效的补丁，这之后才停止了这一做法。

4.3　连锁故障



企业系统的标准架构包括多个网站和网络服务，由功能不同的网站农场或集群组成，农场或集群通过某种负载平衡方式相互连接。如图 4-6，我们通常将一个独立的网站农场看作一层，即便它们实际上并不是一个栈。

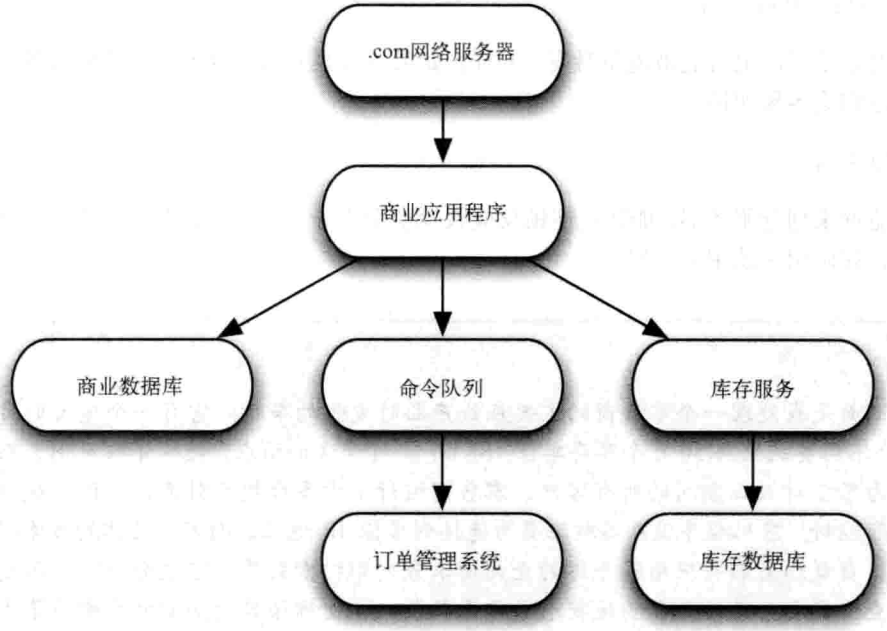


图 4-6　商业系统中常见到的层

在面向服务的架构中，这些看起来不像是传统的层状结构，而更像一个有向非循环图。

系统的故障开始于一个裂缝，这个裂缝通常来自于一些很基本的问题。有各种机制能够延缓或阻止裂缝，不过这是下一章的主题了。没有这些机制，裂缝可能会扩大，甚至会被一些结构性问题放大。发生连锁故障时，一个层中的裂缝会触发其调用层的裂缝。

当某层的故障引发了调用层的故障，就会出现连锁故障。

一个明显的例子是数据库故障。如果整个数据库集群出现故障，那么其他访问数据库的应用程序都将出现某种问题。如果程序处理得糟糕，那么应用层都将开始出现故障。我曾见到一个系统，它会断开任何抛出 SQLException 异常的 JDBC 连接。 每个页面请求都将尝试创建一个新的连接，获得一个 SQLException，尝试推倒连接，得到又一个 SQLException 异常，然后丢给

用户一个栈跟踪。

连锁故障需要某些机制来在层间传输故障，一旦被调用层的不当行为触发了调用层故障条件，故障就跨越了层间的沟壑，扩散开来。

连锁故障常常来自于因低一层的故障而干涸的资源池。没有超时机制的集成点几乎必然导致连锁故障。

如果说集成点是头号裂缝来源的话，连锁故障就是头号裂缝加速器。系统弹性的关键在于防止连锁故障，应对连锁故障最有效的方式是断路器和超时机制。



请记住

- 不要让裂缝跨越了层间的沟壑

当裂缝从一个系统或层跳跃到另一个时，连锁故障就发生了，这通常是由于集成点不够偏执。低层的连锁反应也会导致连锁故障。你的系统肯定会调用其他系统，但请确保在它们崩溃时，你的系统依然存活。

- 审查资源池

连锁故障通常源自资源池，比如因为没有请求返回而枯竭的连接池。获取连接的线程永远阻塞了，所有等待连接的线程也被阻塞了。安全的资源池总是会限制线程等待签出资源的时间。

- 用超时和断路器来防御

连锁故障发生在其他故障之后。断路器通过避免调用出故障的集成点来保护系统。而超时保护可确保调用故障集成点后能够返回。

敲打时间

层跳机制通常以阻塞线程的形式出现，但是我也见过完全相反的情况——异常活跃的线程。在一个案例中，调用层迅速得到一个错误，但是由于历史先例，它假设错误不再发生，只是一个底层的临时错误。有时，底层遭遇竞争条件，于是无可奈何地抛出一个错误。而上层的开发者决定重试这个调用。然而，底层系统并没有提供足够多的信息来区分这是个临时错误还是更严重的问题。结果，一旦底层系统开始出现真正的错误，比如由于切换失败而丢失了数据库的数据包，调用者开始催促得越来越多。底层越是哭喊，上层叫声越大："我让你哭！"敲打得越发起劲。最终，调用层全部的CPU都用于调用底层并记录故障日志。这个时候，断路器确实会帮上大忙。

4.4 用户



用户是可怕的[①]，没有他们，系统会稳定得多。使用系统的人类用户拥有创造性破坏的本领。当你的系统就像悬崖边的车，在灾难的边缘摇摇欲坠时，一些用户就像停落在引擎盖上的海鸥。他落下来了！用户就是有这样的天赋，能在最糟糕的时候做出最糟糕的事情。

> 用户是可怕的。

更糟的是，调用我们系统的其他系统却像终结者的军队一样无情地前进，完全不关心我们多么接近崩溃。

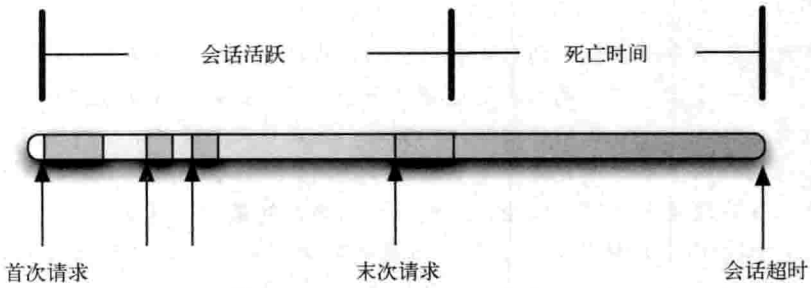
流量

每个用户都在消耗一些系统资源。你系统的容量是有限的，除非你在构建一个类似 BitTorrent 的 P2P 系统。系统的容量取决于你所购买的硬件与带宽的大小，而不是你吸引的用户数量。

流量逐渐增加，终会超过你系统的容量[②]。这时，最大的问题来了：你的系统该如何应对过量的需求呢？

看看 8.1 节对容量的定义。当事务需要很长时间执行时，它意味着系统受到的需求超过了其容量。然而，系统内部还存在一些更为严格的限制。超过这些限制会给系统带来裂缝，并且在压力下，裂缝会传播得更快。

其中一个硬性限制是可用内存，特别是在 Java 或 J2EE 系统中。过高的流量会在多方面给内存系统施压。首先，在网络系统中，每个用户都有一个会话。在用户的最后一次请求之后，会话还会在内存中驻留一段时间。每增加一个用户就意味着需要更多内存。



① 很明显，我是开玩笑的。虽然用户确实给稳定性带来诸多风险，但是他们也是我们系统存在的理由。

② 如果流量没有增加，你又得担心其他问题了！

会话死亡期间，仍然占有宝贵的内存。会话不是神奇的“储物袋”[①]。每个放在会话中的对象都存在内存中，占用着本可被用于服务其他用户的空间。

每个用户都耗用了更多内存。

当内存变少时，就会发生各种奇怪的事情。最轻微的问题是抛出 OutOfMemoryError 异常。如果情况真的很糟，那么日志系统甚至可能没法记录出错。例如，Log4j 和 java.util.logging 都会创建对象来表现日志事件。如果没有内存创建日志事件，就没有日志功能了。顺便说一下，除了刮日志文件，这是对于外部监测的一个很大争议。可恢复的低内存状况将会很快演变成严重的系统稳定性问题。实际上，如果你正在进行本地调用，那么低内存会使 malloc 在本地代码中失效，比如在 Type 2 JDBC 驱动程序内。似乎几乎没有本地代码程序员能做好出错检查，因为我曾见过在内存紧缺时由本地调用引起的 JVM 崩溃。

所以，最好的办法就是让会话尽可能地小。例如，在会话中保存整个搜索结果集用于分页就不好。最好是针对每页的结果重新查询搜索引擎。对于会话中的每个对象，考虑它是否以后还会被使用。它可能在接下来的三十分钟无端地占用内存，而危及你的系统。

如果能在内存充分时保存对象，在内存吃紧时自动释放对象就好了。好消息！确实有方法能做到，就是使用 java.lang.ref.SoftReference 对象来保留其他净荷的对象引用。

你可以将占用空间的昂贵对象作为参数来构建 SoftReference 对象，这个 SoftReference 对象就是一个储物袋，能保存净荷以供后用。

```
MagicBean hugeExpensiveResult = ...; SoftReference ref = new
SoftReference(hugeExpensiveResult);

session.setAttribute(EXPENSIVE_BEAN_HOLDER, ref);
```

这不是一个透明的转变。访问该对象的任何 JSP 或 servlet 都会知道，它们是通过一个间接层访问的。如果内存变少，垃圾收集器会回收 SoftReference 对象的净荷，只要没有净荷的硬引用。

```
Reference reference = (Reference)session.getAttribute(EXPENSIVE_BEAN_HOLDER);
MagicBean bean = (MagicBean) reference.get();
```

添加这个间接层的意义是什么呢？内存低的时候，垃圾收集器就可以回收“软可及”对象，即只被 SoftReference 对象引用的对象。图 4-7 中的昂贵对象就是软可及的；而图 4-8 中的昂贵对象则不是软可及的，因为它被 servlet 硬引用，所以是强可及的。

① 不知你有没有玩过《龙与地下城》，储物袋里面比外面看起来大多了。东西保存在里面但几乎不占重量。它能很方便地让主角装上两把刀、一根狼牙棒、全板甲和五十万的金币。

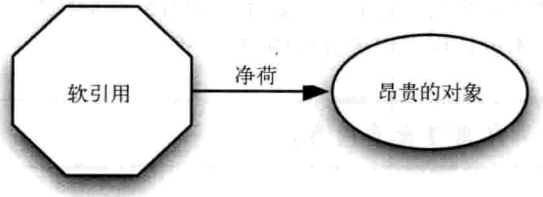


图 4-7　软引用及其净荷

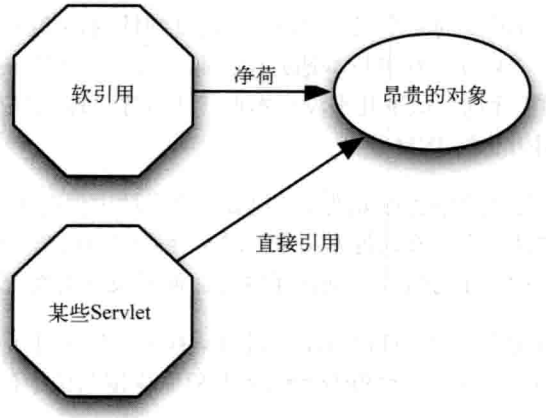


图 4-8　强可及的净荷对象

至于何时回收软可及对象，回收多少，备份多少，完全取决于垃圾收集器。唯一的保证是所有软可及对象会在 `OutOfMemoryError` 异常抛出之前回收。

换句话说，垃圾收集器会尽全力利用你给它的帮助。不过应该注意，回收的是净荷的对象，而不是 `SoftReference` 本身。当净荷被回收了，任何对 `SoftReference.get()`的调用都会返回 null。因此如图 4-9 所示，任何使用净荷对象的代码都要为 null 返回做准备。也可以重新计算昂贵的结果，将用户重定向到其他的活动上，或者采取其他的保护机制。

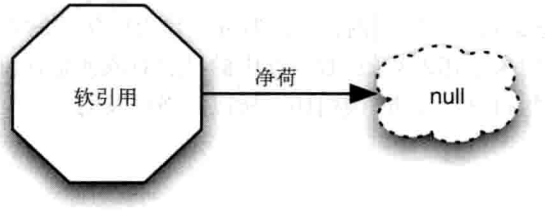


图 4-9　净荷被垃圾回收之后的 SoftReference

在应对不断变化的内存条件时，SoftReference 确实有用，但它也增加了复杂性。通常，最好是将对象排除在会话之外。当你在会话中不可避免要用到庞大或昂贵的对象时，请使用 SoftReference，它能让你在同样内存容量时能服务更多的用户。

昂贵的服务

有些用户比较苛刻。可笑的是，你恰恰想要更多的苛刻用户。例如，对于一个零售系统，那些浏览两个页面、做下搜索然后就离开的用户，可能是数量最多且最容易服务的。它们的内容通常会被缓存（不过得小心使用缓存，详见 10.2 节关于谨慎使用缓存的重要注意事项）。为他们服务的页面通常不涉及外部集成点。你可能会做一些个性化，也许还有些点击流跟踪，仅此而已。

但也有确实想买东西的用户。除非你已经购买了一键结账的专利，否则结账很可能要用四五个页面。这已经和一般用户的整个会话一样多了。最重要的是，结账会涉及一些麻烦的集成点：信用卡授权，销售税计算，地址标准化，库存查询，付运。实际上，购买者增多不止增加了前端系统的稳定性风险，他们也能让后端或下游系统处于风险中。（见反模式 4.8 节“不平衡的容量”。）增加这些对话可能会让营收账面上好看点，但必然会给系统增加很多压力。

没有对昂贵用户的有效防御。他们并非直接的稳定性风险，但是他们产生的额外压力会引发系统其他地方的裂缝。不过，我不建议采取措施避免他们进入系统，因为他们通常能产生收入。那么，该怎么办呢？

对于昂贵的用户你所能做的最好的事情是去积极地测试。确定最昂贵的交易是什么，然后两三倍地提高这些交易的比例。如果你的零售系统期待 2%的转换率（这是零售商的标准），那么你的负载测试应该测试 4%，6%，甚至 10%的转换率。

转换率：真正购物了的网站访问者比例

不受欢迎的用户

如果只用担心那些给我们信用卡号码的用户，那我们所有人的日子都会好过点。然而，如今的现实社会中总在发生奇怪和糟糕的事情，因此我们也必然会碰到奇怪而糟糕的用户。

完全转化

如果一点点好，那么很多一定会更好，对不对？换句话说，为什么不测试 100%的转化率呢？对于稳定性测试来说，这的确是个好主意。但我不会用这个转化率的结果去计划正常生产环境的网络流。根据定义，这是最贵的交易，因此，系统面对的平均压力一定比这个测试中产生的小。构建的这个系统不过是处理最昂贵的交易，而你在硬件上的花费却会多十倍。

他们中有些人并非故意使坏。例如，我曾见过胡乱配置的代理服务器反复请求用户最后的URL。我通过cookie识别用户的会话，然后跟踪会话到注册用户。日志表明用户是合法的。出于某种原因，用户最后一次请求过后15分钟，请求再次出现在日志中。最开始，这些请求每30秒一次，但是不断加快。10分钟后，我们每秒得到四五个请求。这些请求包含用户的识别cookie，但没有他的会话cookie。因此，每次请求创建一个新的会话。它和DDoS攻击极相似，只是它来自于海军基地的一个特定代理服务器。

DDoS：分布式拒绝服务攻击。许多计算机联合起来攻击某个网站，占用网站服务器的带宽、CPU或内存，使网站瘫痪。想象一下格列佛被小人国里的众多小矮人们制伏。

我们再次看到，会话就是网络应用的阿喀琉斯之踵。想摧毁任何动态网络应用程序？从网站中挑出一个深层的连接，然后开始请求它，而不发送cookie。甚至不等回应，发送请求后立即放弃套接字连接。网络服务器从不告诉应用服务器目标用户已经停止等待回应。应用服务器会继续处理请求。它将响应发回网络服务器，而网络服务器将它汇集到队列中。与此同时，100字节的HTTP请求会让应用服务器创建一个会话（它可能占用应用服务器的几千字节）。即便是连接了宽带的台式机也能在应用服务器上生成数十万的会话。

在极端情况下，比如面临海军基地潮水般的会话，你会遇到比高内存占用更糟糕的事情。在我们的案例中，企业用户想知道大多数忠实的客户多长时间又回来了。开发人员写了一个小拦截器，每次将用户的个人配置数据从数据库加载到内存时，拦截器就会更新“上次登录”的时间。但是，在这些会话洪水中，请求包含了用户ID的cookie却不包括会话的cookie。那意味着每次请求被看作一次新的登录，从数据库中加载配置，然后尝试更新“最后登录”的时间。

想象一下，10万笔交易都在更新同一数据库的同一张表的同一行数据，有人一定会陷入死锁。一旦用户配置的锁阻塞了一笔交易（因为有来自不同资源池的连接需求），该行数据对应的所有其他数据库事务就都被阻塞了。很快，这些伪造的登录耗光了所有请求处理线程。一旦这种情况发生，网站就崩溃了。

所以，一种坏用户只是不小心带来了灾害。还有更多的狡猾坏用户，会故意干一些产生不好结果的不正常事情。前者不是故意干坏事，只是无意的，后者就属于明知故犯了。

有一个完全寄生的行业，靠消耗其他公司网站的资源为生。这类公司统称竞争情报公司，它们从你的系统中获取数据，每次一个页面。

这些公司认为，自己的服务和杂货店派人拿着货品清单和笔记板去竞争店收集情报是一样的。但是，其实二者有很大的不同。由于他们请求页面的速度很快，这更像是派了大队人马拿着笔记板去别人的杂货店，他们的人可能会占满过道，而让合法用户无法进入。

更糟的是，这些快速的屏幕搜刮者并不使用会话cookie，所以如果你不使用URL重写来跟踪会话，那么每个新的页面请求都会创建一个新会话。像快闪族一样，很快容量问题会变成稳

定性问题。这一大群“价格侦查员”会拖垮别人的“商店”。

通过使用 robots.txt 文件，避免这类机器人是相当容易的[①]。但请记住，robots.txt 只是你的网站对到访的机器人的一个请求。这个机器人必须请求该文件并尊重你的意愿。这是社会惯例，连标准也算不上，也绝非强制性的。有些站点选择根据 User-Agent 头来重定向机器人和网络爬虫。在最好的情况下，这些代理重定向到产品目录的静态副本，或者站点生成没有价格的页面。（这个思路是允许被大的搜索引擎搜索，但不透漏价格。这样，你能定制价格，提供试用，对国家或市场进行分区测试等。）而最坏的情况下，站点会把这些代理打发到死胡同里去。

会话跟踪

HTTP 是一个奇异的、不太可能的协议。如果你要创造一个协议，以方便艺术、科学、商业、自由演讲、文字、图片、声音和视频，有可能将浩瀚的人类知识和创造力编制进一个单一的网络，那么你不太可能设计出 HTTP。一方面，HTTP 是无状态的。对于服务器，每个新的请求从浓雾中出现并发出命令如 GET /site/index.jsp。一旦回答，它们又消失在浓雾中甚至没说声“谢谢”。如果这些粗鲁而严苛的客户端再次出现，服务器是完全无知的，不知道自己曾见过它们。

网景公司的一些聪明人发现了一种方法，可将额外的位数据移植进协议中。网景起初构思 cookie 数据时，没有什么特别的原因，只是为了将状态从客户端传回服务器，以及从服务器传给客户端。cookie 是个聪明的伎俩。它催生了各种新的应用程序，比如个性化门户和购物网站。然而，具有安全意识的应用程序开发者很快就意识到，未加密的 cookie 数据会被有敌意的客户控制。毕竟，有些浏览器发送用户代理字符串“Mozilla”，但并不表示他就是 Mozilla。（截至第 7 版 Beta 1 的版本，IE 浏览器仍然声称自己是 Mozilla 并且很可能永远会。它的用户代理字符串是“Mozilla/4.0 [compatible; MSIE 7.0b; Windows NT 6.0].”）所以，安全起见，cookie 要么不可能包含真实的数据，要么必须加密。同时，大批量的网站发现在 cookie 中传递真实状态会消耗昂贵的带宽和 CPU 时间。所以应该加密 cookie。

于是，cookie 开始用于小块的数据，只能用持久性 cookie 标记用户或用临时性 cookie 来识别会话。

*会话*是一个抽象概念，令构建应用程序更容易。用户真正发送的是一系列 HTTP 请求。网络服务器收到这些请求，然后通过一系列机制，返回 HTTP 响应。浏览器没有“开始会话”的请求来指出将要开始发送请求，也没有“会话结束”的请求。（网络服务器无法相信会有这样的指示。）

① 参见 http://www.w3.org/TR/html4/appendix/notes.html#h-B.4.1.1。

会话的关键是在内存中缓存数据。早期的 CGI 应用程序不需要会话，因为他们会为每个新的请求启动新的进程（通常是 perl 脚本）。这很管用。没有能像“fork，run，die”一样安全的模型了。然而，为了处理更高的数据量，开发者和供应商转向了长期运行的应用服务器，比如 Java 应用服务器和通过 mod_perl 运行的 perl 程序。这种服务器不会等待每次请求的进程分支，而是始终在运行，等待请求。随着服务器长期运行，你能缓存一个又一个的请求，减少访问数据库的次数。然后，你需要一些方法来识别会话中的请求。Cookie 能干好这件事。

应用程序服务器为你处理所有的 cookie 机制，呈现出类似 map 或 dictionary 的良好编程接口。但是通常，隐形机制麻烦在于，它会因为误用而变坏。当涉及到组装层，这种不可见的机制让 HTTP 看起来像一个真正的应用协议，它确实能严重出错。例如，自制的购物机器人不能妥善地处理会话 cookie。每个请求创建一个新的会话，毫无道理地消耗内存。如果网络服务器被配置对每个 URL 请求应用服务器，而不是在映射上下文中，那么就可能对不存在的页面的请求创建会话。正如你在容量杀手中看到的，控制好会话对可扩展性至关重要。

蜘蛛陷阱

1998 年左右，当 AltaVista 如日中天时，我看到一个开发者的“聪明的”点子让一个网站停掉了。他建立了一个他所称的蜘蛛陷阱。这是一个随机生成的链接页面，链接的文字是根据马尔可夫链生成的貌似合理的词语。网址链接有一个大随机散列值，但这有效地使它们返回到同一个页面。因此，索引器会不断看到指向新链接的不同内容页面。他认为，通过将这些蜘蛛束缚在蜘蛛陷阱中，他会使它们远离网站的其他部分，以保证他的内容被搜索引擎深度链接到。为确保蜘蛛被驻留住，他在每页上都放了 5 个这样的随机链接。

陷阱就像魔法一样，一旦 AltaVista 找到这网站，蜘蛛就不停地请求随机的链接。实际上，它们似乎是多线程运行的，因为我们看到请求的数量呈几何级数增长。问题也在于此。接受这个事实是互联网的绝对法则。搜索引擎总是比你需要更多的带宽。索引器用光了该公司的所有带宽。当我们发现为什么用户在抱怨网站慢的时候，蜘蛛陷阱已经耗费了超过 10 000 美元的宽带费用。这真是一个最精巧的自我 DDoS 的方式。（见 4.6 节。）

我记得他后来写了个商业计划书，将这个马尔可夫链文本生成器卖掉了。

因此，最可能尊重 robots.txt 的正是那些确实可能为你产生流量和收入的机器人，而“吸血虫”会完全忽略它。

我只见过两种解决办法。

第一个是技术方法。一旦你识别出是屏幕搜刮器，就将它阻止在网络之外。如果你使用的是类似 Akamai 的内容分发网络，那么它就能提供这种服务。另外，你可以使用外部防火墙。有些“吸血虫”是诚实的，它们的请求来自真实合法的 IP 地址，使用 ARIN(http://www.arin.net)能帮到你。阻止这些诚实的家伙还是很容易的。

还有些“吸血虫”偷偷掩盖了其源地址，或是从几十个不同的地址发出请求。有些甚至每发送一个请求就改变一次用户代理字符串。当单个 IP 地址声称自己在相同的 5 分钟里在 Windows 上运行了 IE，Mac 上运行了 Opera，Linux 上运行了火狐时，就值得警惕了。确实，它可能是 ISP 级别的大家伙或者运行的一堆虚拟机。但只要这些请求依次爬行整个产品目录，它就更可能是屏幕搜刮器。你最终免不了要阻止一些子网，因此好的做法是周期性地让旧的被阻止的子网过期，这样能让你的防火墙表现更好。这就是断路器的一种方式。

第二种方式是法律性的。为你的网站写一个“使用条款”，声称用户只能以个人或非商业用途查看内容。然后，当屏幕搜刮器开始袭击你的网站时，找律师对他们下手。（很明显，这需要有足够的法律火力来有效地威胁他们。）这些都不是永久性的解决方案。将它看作是病虫害防治吧——一旦停下来，祸患又来了。

恶意用户

最后一组不受欢迎的用户群体是真正的恶意用户。他们*活着*就是为了摧残你的婴儿，能让他们兴奋的也就是摧毁你用血泪和汗水建立起来的网站。我个人认为他们小时候辛苦堆砌的沙雕城堡经常被人踢倒，心灵深处的创伤促使他们对别人也做出同样的事情。

真正有才华的黑客能够分析你的防御，定制攻击，并且神不知鬼不觉地潜入你的系统。你的系统可能由于各种理由被黑客盯上，但这种可能性不大。通常是因为你或多或少惹恼了他们，大多数的恶意用户只是脚本小子，别被这小小名称给骗了。他们纯粹是因为数量众多才具有危害性。虽然被真正的黑客盯上几率很低，但是你的系统很可能已经被脚本小子盯上了。

脚本小子：并不创建自己的攻击，而是下载并利用“真正”黑客所创造的工具。

这本书不准备讲信息安全和网络战争，强健的防御和威慑方法超出了本书的范围。我只将我的讨论限制在安全和稳定性的交叉处，因为它涉及到系统和软件架构。稳定性的主要风险是经典的分布式拒绝服务攻击（DDoS）。攻击者控制网络各处分布的大量计算机对你的网站产生负载[①]。有时候这种负载使用原始 TCP 连接方式，而并不涉及应用层的协议。其他的方式则通过偷偷打破 TCP/IP 协议来攻击你的网络设备。配置良好的现代化网络设备能够防止这种攻击。

① 负载通常来自一个僵尸网络，或受到影响的计算机网络。受感染计算机的守护进程订阅了一个 IRC 通道，僵尸网络的主机可以通过其发送命令。一些脚本小子控制有成千上万个节点，也有传闻称某些僵尸网络的节点多达 100 万个。其中大多是运行老旧过时的 Windows 操作系统的个人电脑。

新的攻击方式是攻击应用程序而非网络设备。它使你的出口带宽饱和，从而对合法用户拒绝服务，并消耗大量的带宽成本。

正如前文所示，基于 J2EE 或 Rails 的网络应用程序最脆弱的方面就是会话管理。当受到 DDoS 攻击时，应用服务器非常脆弱，因此饱和的带宽甚至并非你必须解决的最严重问题。某种专门的断路器能够帮助限制这种危害，并有助于防止太多偶发的流量洪水。

Cisco，Juniper，CheckPoint 和其他的网络厂家都有产品来检测和减轻 DDoS 攻击。对这些产品进行合理配置和检测至关重要。例如，许多管理员设置对单一源 IP 地址每分钟不超过 15 个链接（这是 Cisco 的产品文档举的一个例子）。按照这种标准，每个 AJAX 应用程序都是一个拒绝服务攻击。（我见到一些马大哈的 AJAX 开发人员创造出来的应用被滥用，因此事实可能也差不多就是这样。）



请记住

- 用户消耗内存

每个用户的会话都需要一些内存。尽量减少这样的内存消耗能提高系统的容量。只使用一个会话缓存，在内存吃紧时你可以清空会话的内容。

- 用户做奇怪随机的事情

现实世界中的用户会做些你没法预测甚至没法理解的事情。如果你的应用程序存在一个弱点，他们会通过大量尝试发现它。功能测试的测试脚本有用，但对于稳定测试来说并不有效，雇一群大猩猩来随意敲击键盘更现实一些。

- 恶意用户就在外面

精心设计你的网络能帮助避免攻击。确保你的系统容易打补丁，还要经常打补丁。确保你的框架最新，并让自己持续学习。这年头再让 SQL 注入攻击得手，真是说不过去啊。

- 用户会联手对付你

有时候他们一来可真是人潮汹涌啊。设想 Slashdot 的编辑们朝着你的网站大笑说："大军踏平之！"成群结队的用户就会蜂拥而来，触发挂起、死锁、模糊竞争条件等种种问题。针对访问频繁的深层链接，多运行一些特殊的压力测试吧。



4.5　阻塞的线程

类似 Java 和 Ruby 这样的解释型语言几乎从不会崩溃。当然，它们会有应用程序错误，但很少看见解释器或虚拟机崩溃。我还记得 C 中的一个指针能让整台机器停止工作。（有人还记得 Amiga 机器的 Guru Meditation 错误吗？）但是，解释型语言却有这种问题。应用程序完全死锁，

解释器虽然还在运行，但这没用。

为解决问题增加复杂性可能会带来新的故障模式，可人们却经常这么做。多线程技术为应用服务器带来足够的可扩展性，能够应付最大的网站，但也带来了并发错误。用这些动态语言构建的应用程序出现的最常见故障是 naval-gazing，它是指一个愉快运行的解释器，每个线程都在毫无意义地等待（所谓“等待戈多”）。多线程是如此复杂，需要整本书来诠释它[①]。抛弃“fock，run，die”的执行模型能为程序带来更高的容量，但却引入了新的稳定性风险。

我所处理的大多数系统故障并不是崩溃。进程依然在运行，可是等于什么事都没做，因为处理事务的每个可用线程都死锁了。

在你从连接池中签出资源，缓存或者登记对象，调用外部系统时，线程都可能被阻塞。如果代码结构正确，当两个以上的线程同时尝试访问相同的临界区时，线程可能阻塞。这很正常。假设代码是由熟悉多线程技术的人所写，线程就总是可以解锁然后继续运行。如果这说的是你，那么恭喜，你是那极少数的超一流高手。

这个问题有四部分。

- 想做详尽的测试，不过错误条件和异常会有太多的排列组合。
- 意外的交互可能让以前可以安全执行的代码产生错误。
- 时机很重要。应用程序挂起的几率会随着并发请求的增多而上升。
- 开发人员从不曾让应用程序处理 10 000 个并发请求。

系统不是坏了，而是失灵了

我可能已经试过一百次去解释“系统挂起”和“系统崩溃”的区别。后来我意识到只有工程师被这个问题困扰，就放弃了解释。这就像物理学家试图从量子力学的角度，解释双缝实验中的光子去了哪里。只有一个可观察到的变量是真正重要的——系统是否能完成交易。商业客户将提出这种问题：“这能带来利润吗？”

从用户的角度来看，一个他们无法使用的系统大概和地球上冒着烟的火山口一样。一个简单的事实是，服务器进程在运行并不意味着能帮助用户完成工作，比如购买书籍、查询航班等。

这就是为什么我主张用外部监测补充内部监视器（如擦除日志文件、过程监控和端口监控）。某处的模拟客户端（不在同一个数据中心）可以基于正常情况运行综合交易。该客户端和真正的用户对系统的体验相同，因此无论服务器进程是否在运行，当这个客户端无法处理综合交易时，就是出现了问题。

① 然而，Java 程序员唯一需要的书是 Doug Lea 著的 *Concurrent Programming in Java* [Lea00]。

综合来看，这些条件意味着很难在开发过程中发现程序挂起。你不能依赖于“在系统外测试它们”，最好的提高方式是改善你的代码。在已知模式中使用小部分的原语。最好是下载一个精心设计、使用成熟的代码库[①]。

顺便说一下，还有一个我反对人们从头自己写连接池类的理由。做一个可靠、安全、高性能的连接池比你想象的困难得多。如果你曾为并发安全编写单元测试，就会知道保持池的安全有多难。一旦你想公开度量信息，如在 17.2 节“透明度设计”中讨论的，那么自己造连接池的轮子就不再是有趣的学习，而变为乏味的折磨。

不要信任对领域对象的同步方法。

如果你发现自己在对领域对象写同步的方法，那就需要重新设计。找到一种方法，能让每个线程获得出现问题的对象副本。这很重要，因为如果使用同步方法来保证数据完整性，你的程序就会在多个服务器上崩溃。如果还有台服务器在更改数据，则内存一致也无济于事。另外，如果请求处理线程彼此不阻塞，应用程序的扩展性会更好。

封锁现场

你能找出下面代码中的阻塞调用吗？

```
String key = (String)request.getParameter(PARAM_ITEM_SKU);
Availability avl = globalObjectCache.get(key);
```

你可能怀疑 `globalObjectCache`。你的猜测可能是正确的，但是，没有调用代码告诉你这些调用阻塞而其他的不是。事实上，`globalObjectCache` 实现的接口并没有关于同步的说明。

在 Java 中，有可能子类的方法是同步的，但是在基类或接口的定义里是非同步的。面向对象的纯粹主义者会告诉你，这违反了里氏替换原则。他们说得没错。

你不能用同步的子类实例透明地替换掉基类的。这看起来很挑剔，但是至关重要。

`GlobalObjectCache` 接口的基本实现是一个较直接的对象注册：

```
public synchronized Object get(String id) {
  Object obj = items.get(id);
  if(obj == null) {
    obj = create(id);
    items.put(id, obj);
  }

  return obj;
}
```

① 如果你使用的是 Java 5，而没有使用 java.util.concurrent 中的原语，那可真不应该。如果你还没有使用 Java 5，那么请从 http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html 下载 util.concurrent 库。

同步和里氏替换原则

在对象理论中，里氏替换原则（见 *Family Values: A Behavioral Notion of Subtyping* [LW93]）是指任何 T 类对象的属性为真，则 T 的子类的对象属性也要为真。换句话说，如果基类的方法没副作用，则派生类的也没副作用。一个方法如果在基类中会抛出异常 E，在它的派生类的方法中也应该抛出异常 E（或者 E 的子类）。

Java 并不允许替换原则受到其他声明侵犯。目前尚不清楚在子类中添加同步的能力是里氏原则故意的缺陷还是一个疏忽。

当在一个方法中看到“synchronized”关键字，你心里应该警醒。线程在执行这个方法时，其他的调用者都会阻塞。这种情况下，同步这个方法是正确的[①]。它执行速度快，即便线程间在进入这个方法时有些竞争，它们也都应该执行快。（但是，一点忠告就是，如果每个事务都经常使用它，那么 `GlobalObjectCache` 可能会成为容量限制。见 9.1 节“资源池竞争”，其中有一个例子介绍了阻塞请求对于吞吐量的影响。）

系统的一部分需要检查物品是否有库存，需要向远程系统查询存货状况，这个操作代价很高。这样的外部调用要花几秒钟执行，根据库存系统的工作方式，知道查询结果在至少 15 分钟内是有效的。因为近 25%的库存查询是针对每周的“热门商品”的，最坏情况下会有近 4000 个并发连接向疲劳不堪的库存系统发送请求。开发者决定把这些商品的查询结果缓存起来。

开发者认为正确的做法是通读缓存。一旦命中，就返回缓存的对象；没命中，则执行查询，缓存结果，然后返回。遵守好的面向对象的原则，开发者决定建立一个 `GlobalObjectCache` 的扩展，重载 `create()`方法来执行远程调用。这是教科书般的设计。新的 `RemoteAvailabilityCache` 是一个缓存代理，如 *Pattern Languages of Program Design 2* [VCK96]第 111 ~ 112 页中描述的一般。设置还设计了一个缓存条目的时间戳，当数据太陈旧时让数据过期。这个设计很优雅，但是还不够。

这个设计问题与功能行为无关。功能上看，`RemoteAvailabilityCache` 工作得很好。然而，面对压力时，它会表现很糟糕。库存系统很小（见 4.8 节“不平衡的容量”），因此当前段繁忙时，后端会充斥着请求，最终会崩溃。这时，任何调用 `RemoteAvailabilityCache.get()`的线程都会阻塞，因为一个线程在 `create()`调用里，等待着永远不会返回的请求。

没有人设计了这种故障模式，但设计时也没人避免它。

① 有些 Java 程序员可能已经听说过双重检查锁，是用来避免同步整个方法的。不幸的是，它也不起作用。参见网址 http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html，完整地解释了为什么它不起作用，以及为何所有的修复模式尝试也不起作用。

这个例子显示了这些反模式如何加速裂缝的扩大。故障是由于出现了阻塞的线程和不平衡的容量。集成点处缺乏超时机制会对一个层产生故障，然后成为一个连锁故障，最终导致整个网站崩溃。

显然，如果你问客户“如果网站不能检查货物库存情况时，应该崩溃吗”，客户会笑掉大牙。如果你问架构师或开发者：“假使不能检查库存，网站会崩溃吗？”他们肯定说不会。即便RemoteAvailabilityCache的开发者也没料到当库存系统停止响应时，网站宕掉了。没人将这个故障模式设计在组合系统中，压根儿就没有人考虑这个问题。

第三方库

第三方库导致线程阻塞，由来已久。讽刺的是，企业级软件的客户端库通常在库内实现自己的资源池。这种做法在遇到问题时通常让请求线程永远阻塞。当然，这也绝不允许你配置它们的故障模式，比如所有连接在等待永远不会到来的回应时，该做些什么。

对于这些库你面临的第一个问题是了解其行为。我建议你写一些小的测试用来尝试让库崩溃。使测试用例连接到真实的测试工具上（见 5.7 节），然后看厂商的库是否会调用越来越多的线程。尝试并行发出相同的查询 20 次，看看会发生什么。如果厂商的库做了自己的连接池，那么一旦请求数量超过连接池的大小，吞吐量就会立即下降。（见 9.1 节。）你可能在供应商库中产生了死锁，然而这是宇宙中最悲伤的事情，你没法证明它不死锁。即便你在测试中不能导致死锁，它也依然是脆弱的。

如果库很容易崩溃，你需要对你的请求处理线程作出保护。如果库允许你设置超时，请使用。否则，你不得不采用些复杂的结构，比如厂商库之外的工作线程池，这样请求处理线程能请求执行危险的操作。如果调用能及时通过库进行，那么工作线程和原始请求处理线程会相遇在结果对象。如果调用不能及时完成，请求处理线程就会放弃调用，即便工作线程最终会完成。一旦你踏上这条路，请千万当心。你需要精通并发编程、线程池和编程语言的具体线程模型才行，而且这绝不是个轻松的活儿。

还是花些时间让你的供应商建一个更好的客户端库吧，最好别走这条路。

阻塞的线程经常出现在结合点附近。它们会很快导致连锁反应。阻塞的线程和慢反应能创造积极的反馈循环，将一个个小问题放大成彻头彻尾的故障。



请记住

- 阻塞线程是大多数故障的原因

应用程序故障几乎总是以各种方式卷入阻塞线程，包括曾经流行的“逐步缓慢”和“挂起服务器”故障。阻塞的线程导致了连锁效应和连锁故障。

- 审查资源池

阻塞线程和连锁故障一样，通常和资源池相关，特别是数据库连接池。数据库的一个死锁可能会导致连接永远丢失，并且可能被不正确的异常处理了。

- 使用成熟的原语

学会并使用安全原语。似乎很容易自己写生产/消费者队列，但其实不然。任何并发库比自己造的队列都更经得起测试。

- 用超时防御

你不能证明你的代码中没有死锁，但是你能保证任何死锁都不会永远持续下去。避免使用 Java 的无限 `wait()`方法，请使用有超时参数的方法。应该总是使用超时，即便你会捕获 `InterruptedException` 异常。

- 当心你看不见的代码

所有的问题都可能潜伏在第三方库的代码之中。请十分小心，一定要自己测试一下。只要可能，就应该把给你带来惊喜和故障的代码都拿来仔细调研一番[①]。

4.6 自我否定攻击



自我否定偶尔可以看作是一个人的美德，但绝不是系统的优点。自我否定攻击是指系统或包括人在内的扩展系统，与自己作对。

自我否定攻击的典型例子，是市场部门向“选定的用户”发送包含了优惠信息或特殊报价的邮件。这些东西复制的速度比库尔尼科娃木马病毒（老派的读者也许没听过，就是莫里斯蠕虫）还快。为 10 000 个用户准备的特殊报价准保能吸引来一百万人。网络社区那些四处搜寻低价打折信息的人，能够在数毫秒内就找到并共享这个优惠券代码。

一个著名案例是在 Xbox 360 准备预售时。很明显，美国市场的需求会远远超过供应，当一个大型电子零售商群发邮件推广预购时，写明了预购的确切日期和时间。这封邮件在同一天命中了 FatWallet，TechBargins 等几个大型打折信息网站。它还包括了一个意外绕过 Akamai 的深层链接，可以保证从原始服务器下载每个图片、Javascript 文件和样式。

在预售时间的前一分钟，整个网站都沸腾了，然后就黑了。仅仅用了 60 秒。

亚马逊也曾遭遇到和 Xbox 360 一样的麻烦。2006 年 11 月，亚马逊决定低价促销 1000 件物品，报价只有 100 美元。促销的消息广为流传。结果这 1000 件物品在 5 分钟内就销售一空。不幸的是，这段时间其他物品没有卖出一件，因为上百万的访客都在反复刷新特供页面，期待自己

① 你可能也因此更喜欢开源程序库。我也是。

买到便宜货。

很明显，亚马逊没有为这次特惠准备一个专用的服务器集群（见 5.3 节）。这次特惠本想以低价吸引顾客，增加流量，结果反而带来了负面作用，而没有增加销售收入。我只希望亚马逊能因此发现其系统架构的弱点，尽管已是亡羊补牢。

在零售网站工作过的人都有类似的经历。有时优惠券代码被重用上千次，或者有时候定价出错，让一件货物订购的时候和很多其他货物绑定到一起了。正如保罗所说："营销太好了，随时都能杀死你。"

> 好的营销会随时杀死你。

并非每个自伤事故都该归咎于营销部门，尽管可以这么去想。在共享资源的水平层中，一台服务器出故障可能会毁掉其他服务器。例如，在基于 ATG[①]的基础设施中，总有一个锁管理器来负责管理分布式锁，保证缓存的一致性。（任何用分布式缓存更新 RepositoryItem 的服务器，都要先获得锁，然后更新，再释放锁，然后广播该物品的缓存无效了。）锁管理器是一个单件资源。随着网站在水平方向扩容，锁管理器成为了瓶颈，最后成为风险。如果一个受欢迎的物品无意间被修改了（比如因为程序错误），那么就可能导致最终上百台服务器上积攒了上千个请求处理线程，它们都在排队等待对一个物品的写锁定。

你可以通过构建"什么都不共享"的架构来避免系统引入自我否定。（见 4.7 节的边栏。）如果哪里有不切实际的，就用解耦中间件来减少过度需求的影响，或者通过冗余和后端同步协议来使共享资源在水平方向扩展。也可以给系统设计一种降级运行模式（fallback mode），当共享资源不可用或不响应时，使用这种模式。例如，如果锁管理器提供的悲观锁不可用，应用程序可以使用乐观锁。

乐观锁：保存时可以自由修改对象并检测冲突。
悲观锁：在修改前需要锁住对象。悲观锁更安全，但速度较慢，需要更多的协调。

如果你有充裕的时间准备，且使用硬件负载均衡做流量管理，那么可以把基础架构的一部分用来专门处理流量的激增。当然，这只在额外的流量被定向到系统的这部门时才起作用。（想想促销打折或女性内衣时装秀吧。）在这种情况下，即便专用的部分挂掉了，系统其他部分也还能正常运转。

在这种情况下，当专用服务器出故障时，请确保应用快速失败机制。这样一来，至少其他的前端资源，如网络服务器和负载均衡链接，都不会被牵绊，去等待无用或不存在的响应。

① ATG 商业套件是一个竞争的 J2EE 应用服务器和电子商务框架。参见 http://www.atg.com。

至于人为的攻击，训练、学习和交流则是关键。至少，如果你让沟通渠道畅通，就有可能使系统免受流量激增的影响。你甚至可以在不损害系统的前提下帮助他们实现自己的目标。



请记住

- 保持沟通的畅通

自我否定攻击来自你公司的内部，聪明的营销员自残般地引来了快闪族和网络流量高峰。你能协助并引导这些营销措施，同时保护你的系统，前提是你知道将会发生什么。确保没人会发送带有深层链接的大量邮件。为这些首次优惠创建静态的“着陆区”页面。当心网址中是否嵌入了会话 ID。

- 保护共享资源

编程错误、意想不到的尺度效应和共享资源都会在流量高峰时带来风险。前端负载的增加会让后端处理压力呈指数级上升。

- 大幅度优惠信息会一传十、十传百

以为特价打折信息只限在有限范围内传播就大错特错了，一定会自讨苦吃。这世上没有不透风的墙。即使你限制了能兑换打折福利的人数，仍会有大量群众前赴后继地涌来，梦想着能够花 99 美元买到 Xbox 360[①]。

4.7 尺度效应



生物学上，平方立方定律告诉我们为啥我们没见过大象一般大的蜘蛛。昆虫的重量和它的体积成比例，因此是 $O(n^3)$。腿的长度和覆盖面积成比例，因此是 $O(n^2)$。如果你让小动物 10 倍大，这会让强度重量比变为 1/10，那么腿就不能支撑它了。

我们总是遭遇尺度效应。如果有“多对一”或者“多对少”的关系时，当一方增加时，你就可能会碰上尺度效应。例如，如果一台数据库服务器在有 2 台应用服务器调用时，可能运行良好；但是当再增加 8 台应用服务器时，它却可能崩溃。

开发环境下，每个应用程序就像一台服务器；在测试时，每个系统就像那一两台服务器；但是当面临真实的生产环境时，很多应用程序实在是太小了，而有些程序则是大中型的甚至是巨型的。因为开发与测试环境很少像生产环境有那么大的规模，因此很难看出尺度效应在哪儿困扰你。

① 如果你在 Xbox 360 降价到 99 美元后才读到这本书，那么请理解为“次世代”游戏机。

点对点通信

尺度效应困扰你最严重之处就是点对点通信。应用服务器间的点对点通信很可能当只有一两个实例通信时工作良好，如图 4-10 所示。

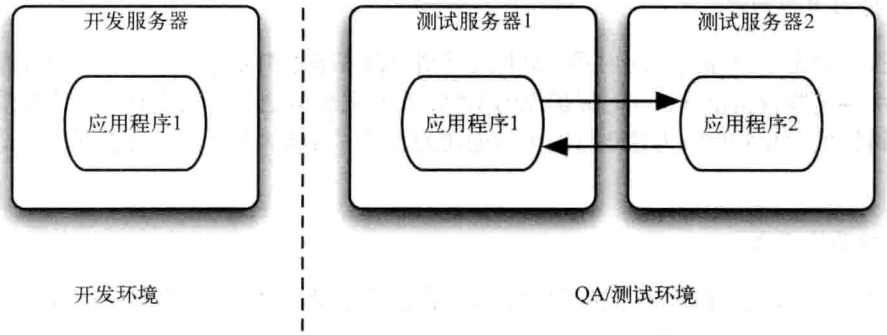


图 4-10　开发与测试中的点对点通信

如图 4-11，用点对点连接，每个实例都必须直接和其他所有实例通话。连接总数会随着实例个数的平方增长。当规模达到上百实例时，连接规模就很惊人了。这是由于应用实例的乘数效应驱动。根据你的系统的大小，$O(n^2)$的规模扩展也许不错。无论哪种方式，你都需要在系统投入生产环境之前知道其规模效应。

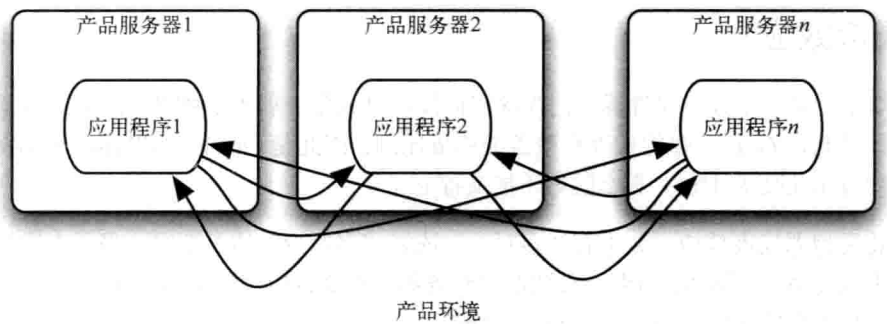


图 4-11　产品发布中的点对点通信

不幸的是，除非你在微软或者谷歌，否则你不可能建立一个和生产环境同等规模的测试农场。这样的缺陷不能测试出，它必须被着意设计出来。

没有“最好的”选择，只有特定场景的最好选择。如果应用程序只有两个服务器，则点对点通信能运行良好[①]。随着服务器增加，就需要不同的通信策略。根据你的基础设施，你能用下列

① 只要通信部分这样做，在另一台服务器死掉时，它就不会阻塞。（参见 4.5 节。）

方式替代点对点通信：

- UDP广播
- TCP或UDP组播
- 发布/订阅消息
- 消息队列

广播能做这工作，但是不节省带宽。它们也会给那些对这些消息不感兴趣的服务器产生额外负担，因为服务器的网卡会收到广播然后通知TCP/IP协议栈。组播更有效，因为他们只让感兴趣的服务器接收消息。发布/订阅消息更好，因为服务器能取得一个消息，即便在消息发送时它并没有监听。当然，发布/消息模式也会给基础设施带来严重的消耗。这很符合极限编程原则：做能起作用的最简单的事。

共享资源

另一种威胁稳定性的尺度效应是“共享资源”效应。通常伪装在面向服务架构或“公共服务”项目中，共享资源是水平扩展层成员都需要使用的机制。对于某些应用服务器，共享资源是集群管理器或锁管理器。当共享资源过载，它会成为限制容量的瓶颈（见8.1节“定义容量”）。图4-12会告诉你调用者如何可能损害共享资源。

当共享资源多余并非排他时，意味着它能一次服务于多个消费者。如果饱和了，可能会增加更多的扩展瓶颈。

但是，很多时候客户端在处理一些工作单元时，共享资源会分配为排他性使用。这种情况下，竞争的规模与同层处理的交易规模成正比。（见9.1节的反模式“资源池竞争”关于竞争对吞吐量的影响。）当共享资源饱和，你多了一个连接积压。当挤压数超过了监听队列，你的交易就失败了。这时，几乎任何事都能发生。它取决于调用者需要共享资源所提供的功能。特别是缓存管理器（为分布式缓存提供一致性），失败的交易会导致陈旧的数据，损害数据完整性。

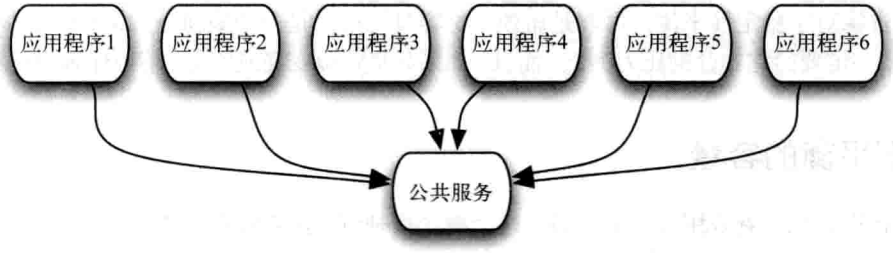


图4-12 多对一依赖

“无共享”

最具可扩展性的架构是**无共享架构**。每个服务器独立运作，不需要协调或调用任何集中服务。在无共享架构中，容量几乎是随服务器数量线性增加的。

无共享架构的麻烦是，它的扩展良好是以失效备援为代价的。例如会话失效备援，用户的会话驻留在应用服务器内存中，当服务器宕机了，用户的另一个请求会重定向到下一台服务器。这需要原始应用服务器与其他设备进行协调。比如引用服务器将用户的会话发送给会话备用服务器。也可能它将会话序列化存储到数据库中，或者共享会话给其他指定的应用服务器。会话的失效备援有许多策略，但都需要将用户会话转移出原始服务器。大多数时候，这也意味着一些共享资源。

通过减少共享资源，也就是减少调用共享资源的服务器数量，可以接近于使用无共享架构。以会话失效备援为例，你能通过制定应用服务器对，让两个服务器彼此做备援服务器。



请记住

- 通过测试环境检查产品以定位尺度效应

当从小型的一对一开发和测试环境转到全规模的产品环境时，你可能被尺度效应所伤。小型环境或一对一环境运行良好的模式，在生产环境中可能效率低下，甚至完全崩溃。

- 当心点对点通信

点对点通信的扩容效果很差，因为连接数量会随着参与系统的数量的平方增加。当使用点对点通信时，请考虑你的系统规模会到多大，也许是足够的。一旦你要处理几十台服务器，就很可能需要用某种一对多的通信来替换它。

- 当心共享资源

共享资源对于稳定性来说，可能是瓶颈，是容量约束，是一种威胁。如果系统必须使用某种共享资源，一定要给它充分的压力测试。而且，如果共享资源变慢或锁定，请确保客户端得以运行。

4.8 不平衡的容量



商业刊物上充斥着效用计算的故事。这个概念很神奇：随着你系统需求的变化，它会自动获得更多的CPU、内存和I/O资源。谁分配给它的？基础设施中有一些实体监视着你程序的性能，当性能不能满足需要的服务水平时会给它更多的资源。这种“主控程序”隐藏在后台，测量你的系统性能并动态重新分配资源。它保证你不再有资源危机。当然，它也会为你使用的资源收费，

就如同你付水费、下水道维护费和电费一样。听起来挺棒的，商业杂志与厂商你唱我和，觉得他们能这样卖掉很多产品。真正实用的效用计算中心今后会出现，但现在我们仅仅只有一些粗糙的近似产品。

在世界其他的99.9%的地方，产品系统都部署在相对固定的资源上。应用程序运行在操作系统上，操作系统跑在硬件上[①]。硬件包括网络接口，上面插上网线。网线的另一端插在交换机上。在传统的数据中心，为产品系统扩大容量需要几星期时间。（验证硬件要求；检查端口可用，制冷能力，电源容量和机架空间；购买硬件；记录变更；架设网络设备并连线；安装操作系统；更新资产管理数据库；分配 SAN 上的 LUN；配置文件系统；部署应用程序；给集群添加应用。）紧急的时候，能够在几天内完成，特别是如果你可以暂时借用别人的服务器，就可以略过采购和安装阶段了。三年前，我曾见到6个新服务器重新接网线，重新安装，重新配置应用程序，所有这些工作都由一个明星工程师奇迹般地在 36 小时内完成。Todd，我由衷地佩服你。这 6 个服务器挽救了网站发布，将它从一个不折不扣的灾难变成了一个轻微的小事故。

所有这些只为说明，除非在极端情况下，你总会卡在可用的资源数量上。在几小时或几天内为短期的需求高峰添加资源实不可行。虽然在代码发布、调校、优化、网络重构或架构改变时，系统能力可能会改变，但在特定的时间点，它其实是非常静止的。

短期时间内，你的硬件能力是固定的。

在多层系统或依赖于其他应用程序时，这将会产生潜在的故障风险。

图 4-13 中，前端网站有 3000 个请求处理线程可用。在高峰时候，大部分会服务于产品目录或搜索结果页面。一些规模较小的数据会出现在企业“叙述”页面。有些会涉及结账流程。在结账相关的页面中，小部分会查询调度系统当地服务团队能否去客户那里帮助安装。你能做一些数学估算判断一下会有多少个线程能同时调用调度系统。数学并不难，虽然它依赖于统计数字和大量假设，却是众所周知地易于操作。但是，只要调度系统能同时服务足够的请求，你就会认为它已经足够了。

不一定。

假设市场营销要提供仅一天的大件电器免费安装活动，导致了一次自我否定攻击。猛然间，并非只有小部分的前端线程涉及调度查询，而是你会看到 2 倍、4 倍甚至多达 10 倍的线程。事实是前端总有能力搞垮后端，因为二者的能力并不均衡。

① 越来越多的“硬件”可能只是个虚拟机。然而，实际上，为了应对需求变化而添加、删除或迁移虚拟机会比你想象的要慢得多。我所知道的最厉害的运营团队只能每天重新平衡虚拟机，而他们希望减慢这个过程，因为虚拟机迁移正在受到严格的变更控制。

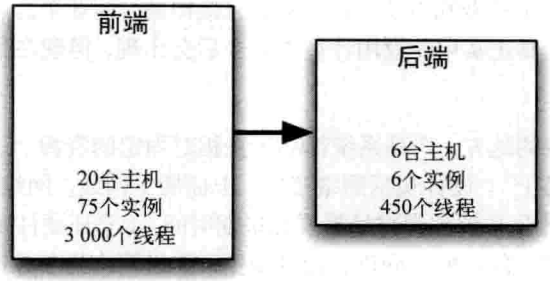


图 4-13 不平衡的容量

要想每个系统能力都均匀匹配是不切实际的，这有多种原因。在这个例子里，将调度系统和网站建得规模相同绝对是资金的误用，只是碰运气哪天能派上用场，基础设施会在五年中每天99%的时间都闲得发慌。

所以，如果不必构建足够大的调度系统以满足前端可能出现的巨大需求，你必须让前端和后端有足够的弹性来应对恐怖的请求。对前端而言，在面临慢响应或连接被拒时，断路器能帮助减轻后端压力。对于后端，握手能通知前端退回请求。还可以考虑使用舱壁，能为后端的其他交易类型保存容量。

通过测试驱除

能力不均衡又是一个在 QA 时很少能被检查到的问题。主要的原因是每个系统的质量测试通常将规模缩小到两台服务器。因此，在集成测试时，用两台服务器代表前端系统，再用两台代表后端系统，产生一比一的比率。

而在生产环境中，由于有大量的预算，这个比率就达到 10 比 1 或更高。

你会在 QA 时，让测试系统与生活环境规模相当吗？这很不错，是吗？当然，你不能这样做。但是，你能利用测试工具。（参见 5.7 节的模式“测试装置”。）通过模拟后端系统在负载下萎蔫的状态，测试装置能帮助验证前端系统是否能优雅地退化降级。

另一方面，你也能检测出后端系统的“正常”工作负载。也就是说，你可以期待现在的需求和交易类型会与以前的负载匹配。如果其他保持不变，那就是一个合理的假设。但是许多因素可以改变你系统要承担的工作量：营销活动；宣传；前端系统的新发布代码；诸如 Slashdot、Fark 和 Digg 一类的“漏斗”网站的链接。作为一个后端系统供应商，你甚至都未必会出现在这些故意产生流量的营销材料中。宣传带来的流量高峰是难以预料的。

所以，如果你的系统会为这些不可预料的调用者服务，你该怎么做？只能考虑周全，凡事都做好准备。首先，使用容量模型确保八九不离十。若是 3000 线程调用 75 个线程，就太不着边际

了。第二，不要只测试你系统的正常负荷。把负荷加倍，直接上最耗费资源的事务，看看会发生什么。如果你的系统弹性良好，它可能会变慢，甚至如果不能在给定时间处理交易，会开始快速失败，但是一旦负载下降，它又恢复了。崩溃、线程挂起、空响应，或无意义的答复，都表明你的系统无法存活，并且可能开始了一系列连锁故障。



请记住

- *检查服务器和线程个数*

在开发和测试阶段，你的系统可能只有一两台服务器，你调用的其他系统的测试版本也一样。在生产环境中，这个比率可能会是 10 比 1，而非 1 比 1。检查前端与后端服务器的比率，以及双方能处理的线程数量，并将生产环境与测试环境作对比。

- *观察尺度效应和用户*

容量不平衡是尺度效应的特例：一方的规模上升比另一方快。季节性、市场驱动或宣传驱动等流量模式的改变可能让通常表现良好的前端系统向后端发送洪水般的请求，就如同 Slashdot 或 Digg 突然对网站产生流量洪水一般。

- *接口两侧的压力*

如果你提供后端系统，就检查一下如果突然给予 10 倍的需求的交易高峰时会怎样。它会完全崩溃吗，还是会放慢然后再恢复？如果你提供前端系统，那么让后端变慢时看看前端会如何。

4.9 慢响应



4.1 节中“基于套结字的协议”介绍过，慢响应比拒绝连接或返回错误更糟糕，特别是在面向服务架构的中层服务环境中。

快速失败能让调用系统快速地完成交易处理。最终的成功或失败取决于应用程序逻辑。另一方面，慢响应会绑架调用系统和被调用系统的资源。

慢响应通常来自过度需求。当所有可用的请求处理程序都已运行，就没有余地处理新的需求了。这可能产生一些潜在的问题。内存泄漏通常表现为慢响应，因为虚拟机越来越难以获得足够的内存空间来处理请求了。这会表现为高 CPU 占用，但是由于垃圾收集，不会作用于事务本身。我曾见过来自网络拥塞导致的慢反应。这在局域网中比较罕见，但是在广域网中，特别是通信协议太冗长，就比较常见了。然而更常见的是，应用程序使套接字的发送缓冲区空，而接收缓冲区满，导致了 TCP 延迟。这通常发生在低级套接字协议中，在接收缓冲区耗尽之前，`read()`方法不工作了。

慢响应往往逐层向上形成连锁故障。

如果你的系统能监视自身性能（见第17章），那么它会告诉你何时不符合服务级别的协议。假设你的系统是一个需要在100毫秒内响应的服务提供者，当过去的20次交易平均时间超过100毫秒，你的系统就会开始拒绝请求。这可能发生在应用层，系统会在指定协议中返回错误。或者，它在连接层拒绝新的套接字连接。当然，任何这样的拒绝必须有文档记录，并让调用者有充分的准备。（由于该系统开发者已经读过这本书，他们会为故障充分准备，让系统优雅地处理这些拒绝。）



- 引发连锁故障的慢响应

上游系统反应慢会使它们自身变慢，在反应时间超过自身超时限制时可能会遭遇稳定性问题。

- 对于网站，慢响应产生更多流量

用户经常点击刷新按钮，这会对已经超负荷的系统产生更多的流量。

- 考虑快速失败

如果你的系统跟踪其自身反应[①]，那么它能告诉你什么时候会变慢。当平均响应时间超过系统允许时立即返回错误（或者至少，当平均响应时间超过调用者超时限制时！）。

- 寻找内存泄漏或资源竞争

数据库连接不充足，争用会带来慢响应。慢响应也会使竞争恶化，导致自我强化周期。内存泄漏对垃圾收集器产生额外的负担，导致慢响应。无效的低级协议可能导致网络延迟，也产生慢响应。

4.10 SLA倒置



SLA（服务水平协议）是一个关于组织必须提供良好服务的合约。它定量地定义了服务，如果服务提供者不能满足这些数据就会遭到处罚。多种趋势导致SLA越来越重要。人力、基础设施和运营等等外包是主要的动力。对IT基础设施库（ITIL，不久将批准为ISO20000）[②]认识的提高和IT服务管理框架（itSMF）[③]也推动了SLA。除此之外，IT业务的总体趋势推高了服务的专业度。IT管理者认为他们自己是重要服务的提供者，为公司的持续运作提供了支持。他们需要量

① 参见第17章。

② 参见 http://www.itil.co.uk/.

③ 参见 http://www.itsmf.com/.

化的 SLA，以便能根据业务需求分配资源，而不是根据大家谁抱怨得厉害。

图 4-14 中，公司的新网站 Project Frammitz 是为高可用性构建的。它的任务至关重要，因此在各级都建立了冗余：电源，网络，存储，服务器硬件，应用程序。它采用无共享架构（见 4.7 节边栏）来尽可能地水平扩展。Frammitz 需要达到 99.99%的可用性 SLA。这意味着每个月只允许略超过 4 分钟的停机时间。



图 4-14 Frammitz 项目架构

尽管仔细规划部署，Frammitz 也只是恰好满足 SLA。

系统本身为高可用性设计，但它还依赖许多其他服务。一个不能访问结算、会计、供货或库存系统的独立网络系统显然是没法卖的。再加上欺诈检测、渠道伙伴集成、关键服务外包、垃圾邮件炮弹、地理编码服务、地址验证和信用卡授权，你真该织一张大网了。这些依赖对于连接的另一端都是 SLA 的弱点。图 4-15 表现了 Frammitz 依赖项，和各自的 SLA。

对于公司内部和外部的每个服务，系统都依赖于传输层的可用性、域名服务和应用层协议。外部连接的这些层都可能失败。除非你系统的每个依赖都提供相同的 SLA，否则你的 SLA 只是这些服务中最差的那个。项目 Frammitz，因为它依赖于合作伙伴 1、定价和促销，这些都没有提供 SLA，所以严格讲，Frammitz 没有提供可用的 SLA。

根据概率计算，情况更糟。如果以最简单直接的方式构建，Frammitz 的故障概率就是任何组件或服务故障的联合概率。也就是说，一个依赖的单点故障足以使 Frammitz 出故障，因此，$P_{\text{frammitz 良好}} = (1 - P_{\text{内部故障}}) * P_{\text{伙伴 1 良好}} * P_{\text{库存良好}} \ldots$。如果 Frammitz 需要 5 个外部服务，每个都达到 99.9% 的可用性，那么 Frammitz 最多只能达到 99.5%。

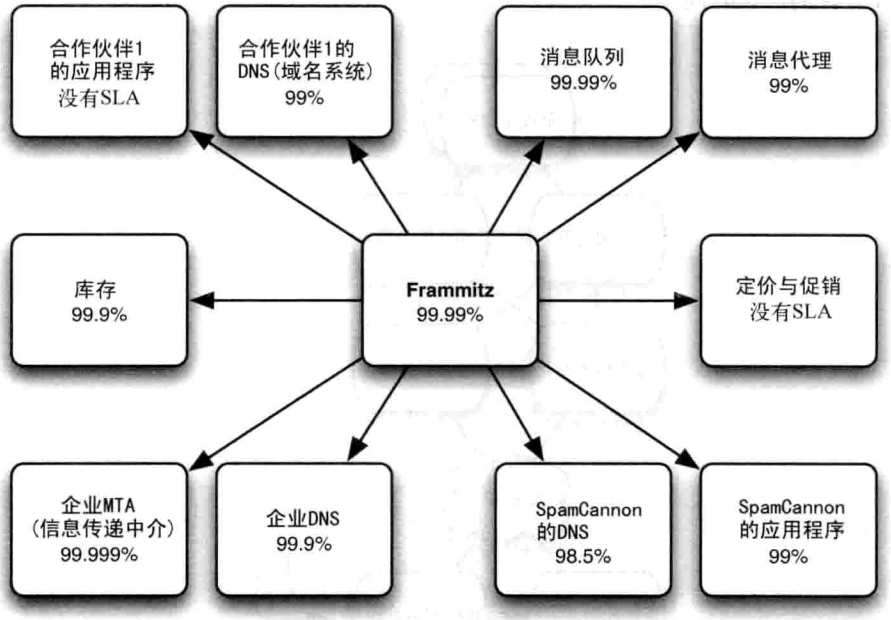


图 4-15 Frammitz 项目的外部依赖

如果 Frammitz 完全和外部系统解耦，那么故障概率就只是 $P_{\text{内部故障}}$。大多数系统会介于两者之间。

调用第三方，服务水平只会下降。

这是一个倒置的 SLA：一个必须满足高可用性 SLA 的系统依赖于低可用性的系统。你必须一厢情愿地承诺高可用性 SLA。对于 SLA 倒置有两个基本反应。首先，你能与较低的 SLA 系统解耦。请确保在没有远端系统时你的程序能持续运行，实现优雅的降级。解耦中间件对于解耦来说是出色的方法，但是却依赖于远端服务的属性，因此不可选。至少，使用断路器能保护你的程序。其次，起草你的服务水平协议时请当心些，不要只简单地声明“99.99%可用度”。（见第 13 章。）

相反，重新对系统的功能可用性定位。不依赖任何外部系统的功能能使你有最大的 SLA。需

要第三方服务的功能只能达到第三方服务供应商提供的 SLA。这是 IT 领域的热力学第二定律[①]：服务水平只能越来越下降。

请记住

- 不要空头承诺

SLA 倒置意味这你在痴心妄想：你承诺了一个只能靠运气才能达到的服务级别。

- 检查每个依赖

SLA 倒置潜伏在意想不到的地方，特别是在网络基础设施中。例如，你公司的 DNS 集群的 SLA 是多少？（我希望它是集群。）SMTP 服务如何？消息队列和代理呢？企业 SAN 呢？SLA 依赖无处不在。

- 解耦你的 SLA

当你依赖的服务崩溃时，请确保系统能维持服务。如果它们失败，你也失败的话，那么你的 SLA 在数学上永远不会比它们多。

4.11 无边界结果集



带着质疑设计，你就能达到伸缩性。问一问“X 系统可能会毁坏我吗？”，然后再设法回避任何来自你盟友的箭。

如果你的应用程序如同大多数一般，很有可能会给予数据库服务器太多的信任。我希望说服你，适当的质疑会帮助你的系统躲过一两个子弹。

常见的代码结构是这样的：发送查询给数据库，然后循环处理结果集的每一行。通常，处理一行意味着给集合添加一个新数据对象。如果突然数据库返回 500 万行而非几百行记录会怎么样？除非你的程序明确限制处理结果的数量，否则它不是耗尽内存就是处理太久。

黑色星期一

你曾经在你的老朋友身上获得过意外的发现吗？比如办公室里最无聊的一个朋友，突然告诉你他喜欢上了蹦极。这种事情恰好发生在我最喜欢的商业服务器上。一天，没有任何警告，网站农场的每个实例（这是 100 多个负载均衡的实例）开始出现异常。似乎是随机发生的。一个实例运行良好，但是几分钟后它占用了 100%的 CPU。三四分钟后，它由于热点内存错误崩溃了。运

① 熵总是增加的。

营团队立即重启它们，但花了几分钟启动并预加载缓存。有时，它们在完成启动前就崩溃了。我们能正常运行的服务器不超过25%。

想象或者回忆一下，如果当时你在场，在早上5点，参加大概20个人的电话会议（没有咖啡），同时调试一个新的故障模式。他们有些人报告了当前的状态，有些对恢复服务设计了一个临时方法，其他人在寻找根源，其中有些人只是在散布假消息。

我们派了一个系统管理员和网络工程师去寻找拒绝服务攻击。我们的数据库管理员报告数据库状态良好，只是负载较重。这是有可能的，因为启动时，每个实例都会发出数百个查询请求来预热缓存。有些实例在开始接受请求前崩溃，这告诉了我崩溃与请求无关。我觉得高CPU占用像是垃圾收集出了毛病，于是我告诉团队开始寻找内存问题。果然，当我在一个实例中观察“可用堆”时，我发现它指向零。不久以后，JVM捕获热点错误。

通常，当JVM用光了内存，就会抛出`OutOfMemoryError`错误。只有在它执行了本地代码并调用`malloc()`出错时才崩溃。我所知道唯一的本地代码是在Type 2 JDBC驱动程序中[①]。Type 2驱动程序用一个薄Java层来调用数据库供应商的原生API。果然，栈转储显示问题是出在数据库驱动中。

但是我们的服务器对数据库做了什么呢？我让数据库管理员跟踪来自应用服务器的查询请求。很快，又有一个实例崩溃了，我们能看看它到底发生了什么，但是查询看上去完全没问题。都是正常的，没有见到奇怪的SQL语句。我看到的最后查询只是查询了一张消息表，服务器用这张表来做数据库支持的JMS实现。这张表不会超过1000行，但是管理员发现它超过了大多数很耗费的查询。

由于某些原因，小数据表会有上千万行。因为应用服务器选择数据库的所有行，每个实例会尝试接受上百万条消息。由于应用服务器提交了“选择更新”查询，会对行加锁。当它试图从消息中构建对象时，会用光内存，最终崩溃。一旦应用服务器崩溃，数据库就会回滚事务，释放锁。然后下一个服务器也会因为查询数据表而崩溃。

我们做了大量手工劳动来弥补应用服务器查询缺乏限制的毛病。到我们让系统稳定时，黑色星期一已经过去了……时间到了星期二。

我们最终发现为何表中有数百万条记录，但这已经是另一段故事了。在访问数据库、分布式对象或Web服务时，这种故障就可能发生。它也可能发生在AJAX请求网络服务器时。

通常形式是在浏览主链接/细节链接时。因为开发环境的数据很少，程序开发者没有遇到这种负面的结果。但是系统运行一年后，每次诸如“获取客户订单”的浏览都可能返回巨大的结果集。

① 对于非Java程序员，本地代码只意味着完全编译的指令。通常，这是动态连接库中的C或C++代码。本地代码在压力下成为崩溃的根源。

一般来说，当调用者允许其他系统决定条款时会发生无界结果集。这是协议的失败。在任何API或协议中，调用者应该说明准备接受多少响应。TCP定义了“窗口”大小。搜索引擎API允许调用者指定返回多少结果，以及起始偏移多少。

标准的SQL语法没有指定结果集限制。对象关系映射工具（比如Hibernate和iBatis）支持查询参数来限制返回的结果，但是通常不限制跟在关系（比如内容的容器）后的结果。因此，要提防任何可能累计无限子查询的关系，比如订单或网站访问的用户配置。审计线索变化的实体也值得怀疑。

如果你是手写SQL语句，使用下面的方法来限制返回的数据行数：

```
-- Microsoft SQL Server
SELECT TOP 15 colspec FROM tablespec

-- Oracle (since 8i)
SELECT colspec FROM tablespec
WHERE rownum <= 15

-- MySQL and PostgreSQL
SELECT colspec FROM tablespec
LIMIT 15
```

一个不完整，但是至少好过没有的解决方案是，查询所有的结果，但是在超过最大行数限制时打破循环。虽然这给应用服务器增加了稳定性，但是它浪费了数据库的处理。

无界结果集是慢响应的常见原因。它们让状态不稳定。

请记住

- *使用实际的数据量*

典型的开发和测试数据集都太小，因此不能暴露问题。你需要给予产品规模的数据集，在查询返回上百万行数据时看看会发生什么。一个附带的好处是，你能从性能测试中获得更多的信息。

- *不要指望数据生产者*

即便你认为这个查询不会有过多结果时，也要当心：由于系统其他部分的原因，它可能没有警告地就超过预期的数据。唯一敏感的数据是“零”“一”和“很多”，因此除非你的查询只是查找一行，否则它就有潜在风险返回结果太多。不要依赖数据生产者只创建了有限的数据，他们迟早会发狂，并且毫无理由地填满数据表，然后那时你该怎么办？

- *给应用层协议加限制*

Web服务调用、RMI、DCOM、XML-RPC，所有这些都可能返回大量的对象集合，从而可能消耗过多的内存。

第5章
稳定性模式

既然你已经看了一些需要避免的反模式，现在我们来看它的反面，我们要来研究上一章里那些杀手级的反模式。这八个健康的模式可以告诉我们如何从架构和设计上降低、消除和减缓裂缝对系统的影响。这里面的模式都不是来帮助你通过 QA 测试的，相反，它会帮助你在软件发布之后，晚上可以睡个好觉，或者说可以和家人共进晚餐而不被打扰。

不要错误地以为采用较多模式的系统就一定比较少的系统优秀，使用模式的多少从来都不是质量的度量指标。相反，我希望你可以养成一种基于恢复的心智模式。别嫌我啰嗦，我在这里再强调一次：请预期系统会失效。明智地使用这些模式可以减轻单个失效带来的破坏。

5.1 使用超时



早期，网络的问题只影响底层软件开发的程序员，他们开发操作系统、网络协议、远程文件系统等系统。而今天，除了一些很小的软件，大部分软件都要不同程度地和网络打交道，这就免不了要遇到一个基本问题：网络会出错。网络的物理连接会断掉，通信中经过的路由器和交换机会坏掉，和你通信的计算机也会坏掉。即使你已经建立了通信连接，这些部件仍然会随时出错。当错误发生的时候，你的代码不能无限期地等待一个永远无法到来的回应，你迟早得放弃等待，希望不出现异常并不是一个好的设计思路。

> 网络永远是不可靠的。

超时是让你放弃等待你认为不会到来的响应的一种简单机制。

我以前曾经做过一个将 BSD 套接字库移植到大型主机的项目。我带来一大堆 RFC 以及 Unix 系统 V R4 的代码来开发这个项目。在整个项目过程中，两件事情让我很困扰。首先，针对不同架构而大量使用`#ifdef` 分隔的代码块，让人觉得这不是一个可移植的单一操作系统，而是二十

个不同的操作系统糅合在一起。其次，网络部分的代码被针对不同的超时而写的出错处理函数弄得千疮百孔。在那个项目结束的时候，我成长了，开始认识和感激那些超时的意义。

RFC：互联网工程任务组（IETF）发布的标准。

设置合理的超时可以隔离故障，在其他系统、子系统和设备中的问题不会成为你的问题。不幸的是，在更高层的抽象里面，远离了肮脏的硬件底层，设置合理的超时机制很少看到了。实际上，很多高层的 API 很少或者没有明确的超时设置。厂家提供的函数库也因为没有超时机制而臭名昭著，这些函数通常为应用程序处理套接字通信，它们屏蔽了实际通信的套接字，也不允许应用程序设置超时。

在单一的应用程序内部超时也是有意义的。一切资源池都会消耗尽，常规的用法要求在有一个资源登记可用之前，调用的线程要阻塞。（见 4.5 节。）

小乔爱问……

所有这些乱七八糟的真的都有必要吗？

你可能会想，就像我在移植套接字库时所做的，处理所有可能的超时会在你的代码中增加不必要的复杂性。这的确会增加复杂性，你可能会发现你的一半代码都是用来处理错误而不是提供功能。然而我认为，目标的本质是生产而不是QA，这是要承受住厄运的打击。这些错误处理代码，如果做得好，会给系统增加韧性。但用户可能并不会因此感谢你，因为没有人会注意到系统未崩溃，但你晚上会睡得更好。

非常重要的是，所有阻塞线程的资源池都应设置一个超时，以确保不管资源最后是否可用，线程都可以被解除阻塞。

请小心使用 `java.lang.Object.wait()`，使用有超时参数 `timeout` 的那个函数形式，而不是使用简单的没有参数的那个形式。对于 `java.util.concurrent` 函数库[①]，也是同样的道理，请使用 `poll()`、`offer()`、`tryLock()`等函数的有超时参数的那个形式。如果不这样做，那么你就可能会永远等待了。

处理无处不在的超时的一个好方法是，把那些要长期运行的操作变成一些原语的集合，这样可以在不同的场合重用。比如，你需要从一个资源池里面得到一个数据库连接，运行查询，将 `ResultSet` 转换成为对象，然后将数据库连接归还给资源池。在这个交互中，至少有三个地方可能会无限期地挂起。与其将这个交互的过程在不同的地方编码实现，同时处理超时错误（更不用说还需要处理其他错误），不如将这一部分变化的交互创建一个 `QueryObject` 来表示。（见《企

① 在 Java 5 中添加。在 Java 的早期版本中，你可以下载 util-concurrent 库。需要适当地替代包名称。

业应用架构模式》[Fow03]）。

使用一个通用的 Gateway 来作为连接建立、错误处理、查询执行、结果处理的模板（见 Spring 框架里面的 JdbcTemplate[①]）。把这个交互的模式包装成一个类，这样也可以更好地对它实行断路器（Circuit Breaker）的模式。

可以看到出错处理经常和重试形影相随。在“尽力而为”的理念下，软件通常会在一个操作超时后去重试。在操作失效后马上重试会有一系列的结果，但只有一部分是好的。如果操作失败是因为一些重大的问题而引起的，那么很有可能重试仍然会失败。一些暂时的失效也许可以通过重试修复（比如，在无线广域网上面的丢包）。然而在数据中心内部，失效也许是由于连接的另外一端引起的。尽管 Cisco 的广告自称什么“自愈网络”，但我的经验是，网络上或者是其他服务器的问题一般都会持续一段时间，马上去重试非常可能失败。

对于客户来说，需要长时间等待是很糟糕的一件事情。如果你因为超时而无法完成一个操作，最好请返回一些结果。它可以是失败、成功，或者通知说你把工作提交排队，等待执行了（具体是什么都不重要）。不管怎样，要返回一个答案。在你重试操作的时候让我等待，可能会使你的响应时间超出我可以承受的超时时间。

另一方面，把工作排队等待后缓慢重试是一个很好的方法，这样可以使整个系统更加健壮。设想，在发送者和接收者之间的所有邮件服务器都需要在线，并且可以随时处理到来的邮件，在六十秒内响应才可以让邮件成功通过。显然，相比起来，存储和转发的方法更加合理。当远程的服务器失效的时候，缓存和重试可以保证在远程服务器恢复后，整个系统就可以恢复。这样，整个工作不会因为一个大系统中的一部分失效而丢失。那么，多快算是快呢？这取决于你的应用和你的客户。对一个面向服务的架构的网站来说，“足够快”也许是小于 250ms 的时间。超出这个数值，你就要丢失处理能力和客户了。

超时很自然地是和断路器协同工作的。断路器可以把超时排成表格，如果太多超时发生，就变成跳闸状态。

超时和快速失效都用来解决延时的问题。超时模式在你需要保护自己的系统不被其他系统的失效而影响时有用，快速失效在你需要报告给外界你无法处理一些事务时有用。快速失效针对的是进入的请求，而超时处理的是发出去的请求。它们是一个硬币的两面。

超时可以防止客户处理整个结果集，从而解决无界结果集的问题，但这并不是解决这个问题的最有效的方法。我认为这只是一个权宜的解决方案，就不深入讨论下去了。

超时适用于一类通用的问题，它可以帮助系统从未知的事件中恢复过来。

① 参见 http://static.springframework.org/spring/docs/1.2.x/api/org/springframework/jdbc/core/JdbcTemplate.html。

请记住

- 应用到集成点，阻塞线程和慢响应

超时可以避免对于集成点的调用变成阻塞的线程，从而可以避免级联的失效问题。

- 应用到从意外的失败恢复

当一个操作花费的时间太长，有时候我们并不关心为什么……我们只是需要放弃，然后继续前进。超时模式可以让我们实现这个功能。

- 考虑延时重试

很多超时都因为远端的网络或系统出现的问题不能马上解决。在这中间重试会很有可能触及同样的问题，产生另外一个超时，从而让用户等待更长的时间才得到一个错误信息。很多时候，你应该把操作缓存起来，事后再试。

5.2 断路器



曾几何时，当电线开始安装在房间里的时候，很多人就变成了物理学的受害者。有的人会在家里的电路上插满电器，每个电器都吸收一定的电流，当电流通过电阻时，它产生的热量和电流的平方乘以电阻成正比（I^2R）。由于在家里没有超导的电线，这些小电器的连接使墙里的电线发热，接着就发生了火灾。噼噼啪啪，然后，房子就没了。

刚起步的能源工业找到了针对电阻发热的部分解决方案，那就是使用保险丝。保险丝的作用就是不等房间起火自己先熔断。这是一个自己先失效，从而控制整体失效的方法。这个很不错的设备工作起来很棒，但是有两个缺点。第一，保险丝是一次性使用的，这样可能会用完；第二，美国的家用保险丝和铜便士的直径相同。这两个缺点经常导致人们自制些高电流低电阻的保险丝（就是一个 3/4 英寸大小的铜盘）。然后，噼噼啪啪地房子又没了。

家用保险丝的发展如同旋转式拨号电话一样，逐渐不为人们所用。现在，人们用断路器来保护房子不被那些电器烧掉。原理是一样的：检测过度的使用，然后先失效，再打开电路。更抽象点来说，就是断路器可以让子系统（电路）失效（比如，由于短路造成的过度的电流吸收）。等危险过去之后，断路器可以重新设置，恢复整个系统的功能。

你可以在软件中使用类似的技术，把一些危险的操作包装在一个组件中，这个组件在系统不太健康的情况下避免被调用。这和重新尝试是不同的，断路器会避免执行操作而不是重新执行它们。

在通常的闭合状态下，断路器和平常一样执行操作。这些操作可以是对外部系统的调用，也可以是可能引发超时和其他运行错误的内部操作。如果调用成功，没有什么特别的事情发生。如果调用失败了，断路器记录失效的状态。当失效的个数（或者更复杂的情况，失效的频率）超出一定的阈值，断路器会把电路变成“开路”的状态，如图 5-1 所示[①]。当电路是开路的时候，对它的调用马上失败，并不实际执行操作。当一定的时间过后，断路器觉得操作有可能成功的时候，它会进入“半开”的状态。在这个状态下面，对于断路器的下一个调用会允许执行危险的操作。如果调用成功，断路器会重置返回到“闭合”的状态，可以接受更多的例行操作。如果这次尝试失败，断路器会返回到“开路”状态，直到下一个超时结束。

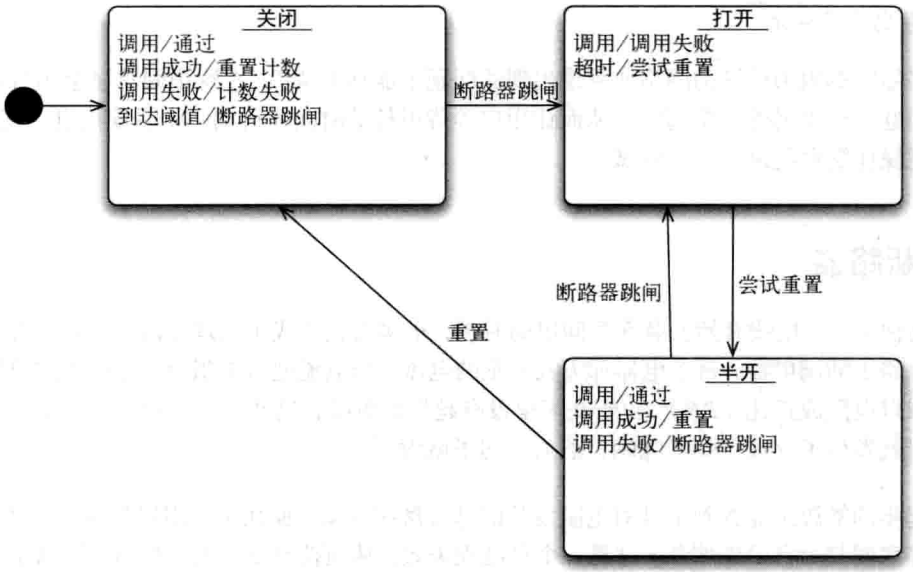


图 5-1　断路器状态跳转

当断路器处于开路状态的时候，所有的调用马上失效。这些可以被某些异常来代表，为了给用户更好的反馈，在开路状态下，可以抛出不同的异常，这样调用代码可以以不同的方法处理它。

基于不同系统的具体特点，断路器可以分别跟踪不同类型的失效。比如，你可能选择对于“调用远程系统超时”设置的下限阈值小于“连接被拒绝”的阈值。

断路器是一种在系统处于压力状态下自动降低功能等级的方法。这可能会对系统的业务有影响。因此，非常重要的是要把系统的干系人包括进来，来决定当处于开路状态时如何处理调用请求。比如，一个零售系统在无法确认用户购买的物品是否有货时就接收订单吗？那如果没法验证用户的信用卡或者交货地址该如何处理呢？当然，这样的对话并非只是在使用断路器时才发生，

① *Pattern Languages of Program Design 2* [VCK96]中的漏桶模式提供了这类计数器的精彩实现。

但是讨论断路器与要求一个需求稳定相比，更适合用来充分讨论这个话题。

系统中的断路器的状态对于另外一群干系人很重要：运营团队。断路器的状态变化应该被记录下来，而且它当前的状态应该要开放出来被查询和监控（详见第 17 章）。实际上，状态随时间变化的频率也是一个系统的有用的度量，它最能指示出企业环境的其他地方发生了问题。

在这种状况下，运营团队需要相应的方法去关闭或者合上断路器。

断路器对于监控集成点、级联失效、处理能力的不匹配以及慢速的响应都很有效。他们可以和超时紧密配合，可以把超时错误和执行错误分开。

请记住

- 如果会受伤就不要调用它

断路器是保护你的系统各种集成点问题的基本模式。当你的集成点出了问题，就不要再调用它。

- 和超时一起使用

断路器在集成点有问题的时候要避免调用。而超时模式一般代表集成点有问题。

- 暴露、监控和报告状态的变化

断路器的弹出一般都代表有严重的问题发生，它应该对运营团队可见[①]。状态的变化需要被报告、记录、分析和关联。

5.3 隔板



船上的隔板是可以焊接的金属板，用来把船分隔成独立的防水的隔室。当舱口被合上后，隔板防止水从一个区域流进另外的区域。这样以来，某处船板穿透之后，整条船不会沉掉。隔板将破坏效果进行了隔离。

你也可以用同样的技术。把你的系统分隔之后，可以避免一个部分的故障破坏一切。物理冗余是最常见的一种隔板。如果有四个独立的服务器，一个服务器的硬件故障不会影响其他的服务器。同样，如果在一个服务器上面有两个应用的实例在运行，其中一个崩溃了，另外一个会继续运行（当然，除非第一个应用是因为某些外部原因崩溃的，这些外部原因也会影响第二个应用）。

给关键客户机预留自己的调用池，从而保护它们。

① 参见第 17 章。

在宏观上说，一个任务关键型服务可以实现为互相独立的几个服务器农场，其中某些是给关键应用预留的，其他的是给非关键应用保留的。比如，一个票务系统可以为客户检票系统单独分配一台服务器，即使其他共享的服务器在处理航班状态查询的请求时不堪重负（在恶劣天气下经常发生），这些保留的服务器也不会受到影响。这样的分隔本可以使我们在第二章里介绍的航空公司，即使渠道合作伙伴无法查询当天航班的价格的时候，仍然可以检票。

在图 5-2 中，Foo 和 Bar 同时使用了企业服务 Baz。因为它们依赖于同一个服务，所以每个系统对于另外的系统都有一些脆弱性的依赖。如果 Foo 突然受到用户的负载冲击，或者因为某些系统缺陷变得行为异常，或者触发了 Baz 中间的一个 bug，Bar 以及他的用户都会受到影响。这种隐含的耦合使得诊断 Bar 的问题（尤其是性能问题）变得非常难。为 Baz 来规划维护窗口也需要同时和 Foo 以及 Bar 协调，有时候也许不容易找到一个对所有客户都适合的时间窗口。

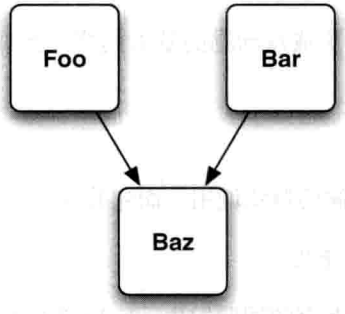


图 5-2　隐藏联系

假设 Foo 和 Bar 都是有严格 SLA 的关键系统，那么如同图 5-3 所示，把 Baz 系统分隔成不同的部分是比较安全的做法。

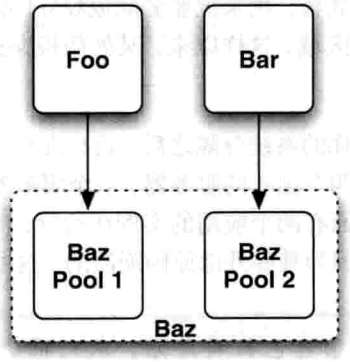


图 5-3　切分系统

隔板和系统容量

为每一个客户保留不同的服务器池，Baz需要更精确的需求预测。Foo和Bar需要更精确地给出他们的需求，因为每个人都有严格的限额资源可用。如果服务器池是组合在一起的，那么一个客户的额外资源要求可以从另外一个客户的过剩的容量里面来补充。这样，整体的系统的容限会提高。

如果Foo和Bar的峰值负载的季节不同，或者甚至一天中的峰值负载时间不同，那么公共的共享资源池所需要的容量要小于单个服务需要的容量。这种使用服务器的方法带来的高效率是IT迁移到面向服务的体系架构的主要动力。那么我们如何在保持这种高效的同时又能够获得隔板带来的安全性呢？

一种可以带来前述两种好处的方法就是虚拟化。Baz可以用同样的物理服务器池来创建专门为Foo和Bar服务的虚拟服务器。那样的话，在客户之间移动负载容量变成了对于虚拟机的管理性改变。启动额外的虚拟机只需要短短的几分钟。最新版本的VMWare ESX甚至能根据服务级别策略自动移动虚拟机。迁移一台活动的虚拟机通常不超过十秒钟。

虚拟服务器提供了一种实现隔板的良好方法。

为某些关键的客户预留一些专用的容量可以移除大部分不同服务和客户之间的隐含联系。他们也许还要共享一个数据库，所以可能会遭受不同实例之间的死锁问题，但这是另外一个反模式了。

当然最好可以保留所有的系统容量。假设失效会发生，但是你需要考虑如何减少由失效引起的损失。这并不简单，也没有一条规则可以适用于所有的情况。相反，你需要检查每一个系统能力的缺失带来的影响，并且把那些影响和系统的体系结构来进行对照。目的是可以确定系统的自然边界，确保你可以分隔系统，这既在技术上可行，在财务上也是有益的。这些分隔的边界会和调用者、功能以及系统的拓扑结构相对应。

从小的范围看，CPU绑定是用隔板分隔系统的一个例子。把一个进程绑定到一个CPU或者CPU组会保证操作系统调度某个进程的线程只在指定的CPU或者CPU组上面执行。这样会降低进程从一个CPU迁移到另外一个CPU时引起的Cache缺失，CPU绑定通常是作为提高性能的方法。如果一个进程出现混乱，开始占有所有的CPU周期，它通常会把整台机器拖垮。我看到过一个八CPU的服务器被一个进程占有。而如果那个进程只是绑定到一个CPU，那么它就只能使用那个CPU上的可用的周期了。

你也可以分隔一个进程中的线程，让不同的线程组去专门处理不同的功能[①]。比如，通常预

① 例如，在第9版之前，WebLogic允许分配一套线程给自定义活动队列来处理特别的活动。（在9.x版，这被称为工作管理器的一个约束。）

留一个线程池来处理管理请求是很有好处的。这样的话，如果所有的请求处理线程都挂起了，系统仍然可以出来管理请求，比如收集数据以做事后分析或者请求关机。

隔板在面对失效的情况下仍然可以维持服务或者是部分服务。它们在面向服务的架构里面特别有效，当其中的一个服务失效后会影响企业的其他业务。实际上，SOA 中的一个服务就是整个企业的一个单点故障。



请记住

- 挽救部分船只

隔板模式把系统分隔，使得坏事情发生之后，系统仍然可以保证部分功能。

- 决定是否要接受不高效的使用资源的方法

当系统不处于危险的时候，分隔服务器代表着每个部分都需要多预留资源。如果所有的服务器都放在池子里面，那么要求全部预留的资源就会减少。

- 选择有用的粒度

你可以选择分隔一个应用中的线程池、服务器上面的 CPU 还是一个集群中的服务器。

- 共享服务模型非常重要的考虑

在面向服务的体系架构中，可能有许多企业应用系统依赖于你的应用。如果你的系统因为连锁反应而停止服务，那是不是整个公司都停止了呢？那么，你需要放置一些隔板。

隔板与性能

把一个多线程的应用服务器绑定到一个 CPU 上面可能会降低它的性能，特别是在单独测试的情况下。当一个应用服务器的多个实例在一个主机上面运行的时候，情况并不是很明朗。当一个应用服务器的现场可以在所有的 CPU 上面调度的时候，上下文切换和缓存刷新会降低整体的性能。优化性能涉及很多方面，它需要平衡正确的应用服务器的实例个数、每个服务器的线程的个数、分配的内存的数量、应用内部的冲突的概率。请确定要测试系统最高的整体性能，而并不只是测试单台服务器的最好响应时间。

5.4　稳定状态



Roget's Thesaurus 第 3 版为单词“瞎摆弄”（fiddling）给出如下的定义：无所事事地、无知地、破坏性地处理某事。它通常也和类似傻瓜、干预、瞎搞、笨拙以及猴子是同义词。瞎摆弄之后通常就是后悔不迭的时刻，顷刻间，你认识到自己敲错了键，把服务器关闭了，删除了重要的

数据，破坏了系统的稳定和谐运行。每次有人去操作服务器的时候，就都有可能引入不可预测的错误[①]。

需要尽可能地让人远离一个上线运行的系统。如果一个系统需要人不停地调整和操作才可以保证正常的运行，那么管理员会养成持续登录系统的习惯。这样，不可避免地会带来“瞎摆弄”。最终，我们需要系统可以在没有人工干预的情况下持续运行。

除非系统每天都会停机（这时要看是否存在稳定性的反模式），大多数需要登录系统的原因是要清除日志或者数据。

> 不要鼓励任何瞎摆弄。系统应该在无人干预的情况下一直运行下去。

任何积累资源的机制（不管是文件系统中的日志文件，数据库中的记录，还是内存中的缓存），就如同中学计算题里经常提到的桶一样。桶以一定的速度随着数据的积累而填满，同时，它也需要以同样或者更大的速度去消耗，不然的话，这个桶会溢出，由此接连不断会发生坏事：服务器停机，数据库变慢抛出错误，响应时间变慢。稳定状态模式要求，对于每一种积累资源的机制，一定要有相应的机制去回收这一资源。你会看到各种可能积累起来的项目以及如何避免瞎摆弄。

数据清除

看起来是一个非常简单的原则，计算资源是有限的，所以你不可以无限制地增加对它的消耗。同时，在匆匆忙忙激动地推出一个新的杀手级应用的时候，而这个应用又被冠以“下一个关键项目”、“公司未来”等名目，数据清除通常被排在了不那么重要的地位。同样，演示数据清除也是不那么令人激动的事情，也许其他任何功能的演示都比数据清除有意思。有些时候，系统可以在真实世界中运行起来，你就会高兴得跟中奖似的。何况运行的时间相当长，积累的数据都无法处理，这样么一个听起来很“高端”的问题，似乎反是你很希望碰到的问题呢。

> 虽然应当如此，但数据清除功能很少被加到第一版中。

不管如何，你的数据库总会长大，当它长成个半大小子的时候，大概相当于人类的两年时间，它就会变得心情烦躁、忧郁和不满。盛怒之下，它开始破坏整个系统（或者开始抱怨没有人理解自己）。

① 我知道的一个事故，一个工程师发现服务器的根磁盘镜像没有同步。于是他执行“resilver”命令，让镜像同步。不幸的是，他敲了个错字，而将根磁盘与新的空驱动器同步了，这个空驱动器是刚刚来替换坏磁盘的，因此立即破坏了服务器上的操作系统。

数据增长的最明显的症状，就是数据库的 I/O 数据请求稳步增加，你也可以看到在稳定负载下增加的延时。

数据清除是讨厌的、细致的工作。保持数据库的引用完整性只是问题的一半，要在清除无用数据的时候不留下孤立记录，是极为困难的。另外一个比较难的问题是在数据移除之后还可以保证应用可以运行。

比如，在数据集合中间的数据丢失后，应用还会继续工作吗（注意，在 Hibernate 框架下面，不会！）。因此，数据清除往往是在第一个版本发布之后还遗漏的功能，这里隐含的逻辑是：我们会在发布后六个月来实现数据清除。（一定程度上，人们喜欢说六个月，正如程序员经常会估计一项工作需要两个礼拜完成一样。）

当然，发布之后，总是要推出许多紧急的版本发布来修改严重的缺陷，或者增加一些市场人员焦急等待的“必须要有”的特性。第一个六个月很快就过去了，当第一个版本发布的时候，保险丝就已经被点亮了。

实践中的数据清除

我在 OTUG[1]上面进行过一个讲座，最终导致了这本书的产生。我非常高兴地看到我的项目组和投资方大都参加了这个会议。在讲到数据清除非常重要而又非常容易被忽略的时候，我看到项目组的每个人都在点头（同时睁大眼睛注视着我）。你可以想象当我们发布了没有数据清除功能的第一个版本时该有多懊恼了。

我们最终实现了一个比较完整的清除的流程，基于测量我们最短的保险丝来决定我们需要设计多长的保险丝。最终，我们发布了第一轮的数据清除功能，它正好可以满足应用，清除最大的数据容量。这样为我们赢得了时间，后续的版本发布了低容量数据的更严格的清除程序。

另外一个你经常碰到的问题是老的日志文件。

日志文件

一周下来，日志文件和一册精算表速查的书一样乏味，很少有人愿意仔细研究它们。多数人会直接把它们扔进垃圾桶。整个上个月的日志文件甚至更糟，它们的唯一作用只是占据宝贵的磁盘空间。

如果不去检查这些文件，那么它们只是一堆无意义的字节。日志文件无限制地增长，最终会把包含它们的文件系统占满，不管是专门存放日志的卷还是根磁盘，或者是应用所安装的目录（但愿不是），这些都会带来问题。日志文件占满文件系统的时候，会破坏系统的稳定性，因为文件

① The Object Technology Users' Group，在明尼苏达州，明尼阿波利斯和圣保罗这对双子城，参见 http://www.otug.org/。

系统被占满的时候会引入各种负面的效应。在 UNIX 系统上面，根据文件系统的配置，一般最后的 5%到 10%的空间是为根用户保留的。这代表当文件系统在 90%到 95%满的时候，应用程序会发生 I/O 错误。当然，如果应用是以根用户权限运行的，它可以用完系统的最后一个字节。不管怎么样，操作系统会把错误报告给应用程序。在用 Java 开发的系统里，会抛出异常 `java.io.IOException`，而.NET 系统会抛出 `System.IO.IOException`。对于 C 的系统，是错误号 ENOSPC（请举手，谁在每次调用完 `write()`函数会检查 `errno`？）。在以上所有的情况下，日志函数库都不会直接处理 I/O 异常；反之，它们会把它包装起来，然后在应用的代码里面抛出一个新的异常[①]。

接下来会发生什么，每个人都可以猜到了。在最好的情况下，日志文件系统是和处理交易的关键数据存储分开的，应用程序代码自我保护得很好，用户不觉得有什么事情不妥。比较不那么让人愉快，但是仍然可以接受的是，系统会打印出一个措辞友好的出错信息，让用户稍微耐心一点，等待系统把一切都处理好之后再回来。不够友好的处理方式是给用户打印一个调用栈信息，让用户自己去判断。

更糟糕的是，我在一个系统上看到开发者在 servlet 处理流程里面增加了一个“通用异常处理程序”。这个处理程序会记录所有的异常，它也是可以重入的，所以如果在记录一个异常的时候又发生了异常，它会同时记录原始的异常和新的异常。当文件系统满的时候，这个可怜的异常处理函数变得发狂了，它会记录下不断增长的一系列异常。因为系统有多个线程，每个都尝试记录自己的异常，所以这个应用服务器会在很短的时间内把八个 UltraSPARC III CPU 占满。这些异常很快地消耗了所有的内存，然后接着就导致了 JVM 的崩溃。

大日志文件还有个不那么严重的问题，是它的信噪比很低。看看网络服务器的日志，除非是用来进行 WebTrends 之类的分析，否则基本上不会在上个月的访问日志中发现什么有价值的东西。假设有八百万的请求，由于各个页面的内容不同，可能对应着 800'000 到 4 000 000 页面视图，Apache 的公共视图格式一天产生大约 1GB 的访问日志。没有人可以在这么大量的数据中找到什么有意义的数据。因此，没有任何理由把这些日志文件留在生产系统里，应该把它们复制到一个中间地带来进行分析。

> 不要把这些日志文件留在生产系统里，应该把它们复制到一个中间地带来进行分析。

当然，最好是首先避免日志把文件系统塞满。只需要花几分钟配置一下，就可以处理好日志文件的归宿。

很多 LOG4J 的移植实现，包括 Log4R（Ruby）以及 Log4Net（任何.NET 语言），都支持一

① Log4J 在这方面是个很好的特例，它使用了可插式的 ErrorHandler 策略来处理任何“追加”的异常。

个 RollingFileAppender，可以用来基于文件大小来循环使用日志文件。你应当总是使用 RollingFileAppender，而不是默认的 FileAppender。在 java.util.logging，默认的 FileHandler 可以通过把 limit 属性设置成为允许写入当前文件的最大字节数来实现基于大小循环使用日志。count 变量控制保留多少老文件。显然，limit 和 count 的乘积决定了日志可以占用多少空间。

对于一些历史遗留代码、第三方代码或者任何不是有很好的日志框架的代码，各种 UNIX 下都有 logrotate 工具可用。在 Windows 下面，可以在 Cygwin 环境来编译 logrotate，或者可以手工写一个.vbs 或者.bat 脚本来做这件事。日志可以很好地帮助实现透明性。确定所有的日志文件都需要循环出去并且最终会被清除，不然，你就得花时间来修复本应帮你修复系统的工具。

数据库中的数据、磁盘上面的日志文件，各种持久化数据有的是办法会阻塞你的系统。如同老广告里面的叮铃声，内存中的垃圾也可以破坏你的应用。

小乔爱问……

《萨班斯法案》是什么，我们需要永远保留日志吗？

你有时候也许听到人们讨论日志的时候提到《萨班斯法案》要求。《萨班斯法案》对于IT基础设施和运行提出了很多严格的要求。其中的一个要求是，公司对于可能产生重要的财务信息的系统应有足够的控制能力。换言之，如果记账系统会提供数据给公司做财务报告，那么公司必须能够证明没有人可以滥用记账系统的数据。

对于很多面向客户的站点，这实际上不太重要，但经常却被认为是必需的。财务数据来自订单管理系统或者信用卡处理系统，而不是网络和应用服务器。网站不可能按照《萨班斯法案》的要求保持几年的访问日志，用DVD和磁带记录都不行。再说，网站访问日志能够证明财务控制无懈可击吗？不可能。跟踪管理员登录会话记录才有可能。

不幸的是，法律的问题从来都不可以通过理性的可能性分析来决定，特别是在像《萨班斯法案》合规这样含混不清的领域。你最好的方法就是和公司的CIO或者合规部门人员一起工作。（很多公司有专门的《萨班斯法案》顾问。）他们可以帮助定义你的系统如何保持合规。尽早开始这些讨论，因为这一讨论牵涉到法律、IT、财务部门，所以不要期待能很快就有结论。

内存中的缓存

10.2 节的模式“谨慎使用缓存”讨论了很多关于缓存的问题。对于一个需要长期运行的服务器，内存就好比是氧气。内存低的状况对于稳定性和系统容量都是一个威胁。所以，在构建任何缓存的时候，必须问以下两个问题：

- ❑ 可能的关键字的数量是有限的还是无限的?
- ❑ 缓存的项目会变化吗?

如果关键字的数量没有上限，那么必须要限制缓存的大小。除非关键字的空间是有限的而且内容是静态的，否则缓存需要设计一些失效机制。你当然可以考察使用最近最少使用（LRU）或者工作集算法，但大部分情况下，周期性的清除就可以满足要求了。

使用缓存不当是内存泄露的主要原因，这又往往会引起每天服务器重启的恐怖事件。没有什么可以让管理员每天跟做家务似地去登录到运营系统。

垃圾的积累是系统响应变慢的主要原因，所以稳定状态可以避免这个反模式的发生。稳定状态同时让系统管理员不必老去登录入运营服务器，从而带来更好的运营方式。



请记住

- 避免瞎摆弄

人工干预会带来问题。减少反复的人工干预。你的系统应在不需要人工清理磁盘和晚上重启的情况下，至少可以连续运行一个开发周期。

- 使用应用逻辑来清除数据

DBA 可以建立脚本来清除数据，但他们通常不知道数据移除的时候程序的行为如何。维护逻辑的一致性，特别是当你使用 ORM 模型，就需要用应用程序来清除自己的数据。

- 限制缓存

内存中的缓存可以提高程序运行的速度，用多了又会降低其速度。限制缓存可以使用的内存数量。

- 循环使用日志

不要保存无大小限制的日志文件，基于大小来设置日志文件的循环使用。如果为了合规而需要保留日志，请保留在非运营的服务器上面。

5.5 快速失效



如果说慢响应比没有响应更糟，那么最糟糕的是一个很慢的失败响应。这就好比是在交管局排过了其长无比的队伍之后，被告知填错了表格，然后又回到队伍的尾部重新排队。利用 CPU 的周期只是用来产生需要丢弃的结果，还有比这更浪费资源的吗?

如果系统可以提前知道某个操作会失败，那么就要让失败尽量快一些。这样的话，调用者就

不需要占有它的资源等待了，它可以进行其他工作。

那么系统如何知道它是否会失败呢？我会展现哪些秘密的启发式规则呢？这是和 Intel 跳转预测算法类似的应用层的等价算法吗？

实际上，我们的方法要世俗得多。有很大一部分的问题是资源不可用的错误。比如，一个负载均衡器收到连接请求的时候，如果发现服务器池中没有一个服务器可用，它应当马上拒绝这个请求。有些配置让负载均衡器把连接请求排队，希望在短时间内，有一个服务器会变为可用。这违背了快速失败的模式。

在事务开始时检查资源的可用性。

在任何面向服务的架构，应用可以根据请求的类型而得知大概需要什么样的数据库连接和外部集成点。这个服务可以快速地得到需要的数据库连接，同时确认围绕集成点的断路器的状态。它可以通知事务管理器来开始一个事务。这和厨师在开始做饭的时候把所有的原料准备好类似。如果有任何一个资源不可用，那么就马上失效，而不是去完成一半的动作。

黑

我为一个图片工作室做了个有趣的项目，涉及修改一个为高清晰度打印而渲染图像的软件。这个软件的老版本有个问题，会为下游的人们生成多余的东西：如果任何一个颜色轮廓、图像、背景或者 Alpha 掩码缺失，它就会渲染出一张黑色的“图像”，即一张像素全部是零的图像。这个黑色的图像会被打印流程处理并且打印出来，不仅浪费纸墨，也浪费了时间。质量检查员会把打印的黑图找出来，送到上游人员那里进行诊断、调试和纠正。最终他们会修正错误（通常是把开发人员叫到打印机前面），重新制作印坏的图像。因为命令已经发出，他们会加快返工的过程，也就是打断正常的工作流水线，把重新制作的任务排到最前面。

我的团队接手这个渲染软件的时候，使用了快速失败的模式。当一个打印作业到达的时候，渲染程序会首先检查每个字体（字体缺失也会引起类似的返工，只是产生的不是黑色图像）、图像、背景以及 Alpha 掩码是否存在。它也会预先分配内存，所以后面不会发生内存分配失败。渲染程序不等这样的失败浪费若干分钟计算时间，就会马上返回给作业控制系统。特别是，无效的作业会从流水线上拿掉，这样就避免了不完整的作业在流程中等待的状况。我们发布新的渲染程序后，软件引起的返工率[①]立刻下降到零。

我们唯一没有预留的就是保存最终图像的磁盘空间，我们在客户的指引下违背了稳定状态的原则，他说他们有很牢固的数据清除流程。最终发现，所谓的数据清除流程只

① 但依然会因为其他的质量问题而返工：相机中的灰尘，曝光不足，裁剪不好，等等。

不过是靠一个人时不时删除一批文件。在我们发布后不到一年的时间里，磁盘空间就占满了。确定无疑的是，我们违背了一个快速失败原则，而正是这个原因导致了浪费不少精力却没有能够报告错误。软件仍旧会去渲染图像，用几分钟的时间来完成计算，然后在日志中抛出一个 IOException 异常。

另外一个在 Web 应用中实现快速失效的方法是在接收到请求的 servlet 或者控制器里，在加载 EJB 或者域对象之前首先检查参数。但是要小心，你不可以违背域对象的封装性。如果检查的不简单是空、非空或数据格式，就应该把这些有效性检查移到域对象的内部或者应用程序去。

即使是实现了快速失效，也应在系统失效（资源不可用）时给出和应用失效（参数无效或者状态无效）不同的错误报告。仅仅报告一个泛泛的“错误”消息，会使上游的系统因为某些用户输入了不合法的数据然后按了 Reload 键三到四次，就闭合了断路器。

快速失效的模式可以通过避免慢速响应而提升系统的整体稳定性。和超时结合，快速失效可以避免发生级联的失效。它也可以帮助系统在部分失效的时候维持一定的处理能力。

请记住

- 避免缓慢响应，尽快失效

当系统无法满足一定的 SLA 的时候，要尽快通知调用者。不要让他们等待一个错误信息，不要让他们一直等到超时，否则你的问题就演变成了他们的问题。

- 预留资源，尽早验证集成点

根据不要做无效工作的主题，确保你在开始之前是可以完成一个事务的。如果关键的资源不可用，比如调用需要的断路器是开路的，那么就不要浪费时间去调用它。它的状态在调用开始和处理事务的中间发生变化的可能性很小。

- 输入的校验

在你预留系统资源之前先进行基本的用户输入验证。不要先检查数据库连接，得到域对象，填充好这个对象，然后调用 validate()函数时才发现，一个必需的参数没有输入。

5.6 握手



握手指设备间规范通信的信令。比如 RS232（现在是 EIA232C）串行通信协议依赖于接收者来指示是否就绪可以接受数据。模拟的调制解调器依赖于握手来协商设备双方都同意的速度以及信号编码方法。同样，如前面介绍的，TCP 使用三次握手来创建一个套接字。TCP 的握手允许接收者通知发送者在它准备好之前不要发送数据。握手在底层的通信协议里面几乎随处可见，可是在应用层几乎看不到。

一个很悲哀的现实是HTTP握手处理得并不好。基于HTTP的协议，比如XML-RPC或者WS-I Basic，都少有选项支持握手。HTTP提供一个回应码“503 Service Unavailable”，按照定义，这代表一个临时的状况[①]。而大多数的客户端却无法区分不同的返回码。如果返回的不是“200 OK”[②]，“403 Authentication Required”或者“302 Found (redirect)”，客户端或许会把这些作为致命的错误。

同样，CORBA、DCOM、Java RMI之下的协议，都不能很好地报告它们是否就绪。握手的全部目的就是要服务器通过限制自己的工作负载来保护自己。服务器需要有一种方法来拒绝进来的请求，而不是变成一个随时处理任何请求的受害者。我可以在基于HTTP的服务器上实现的最接近的一种方法，是依赖于网络或者应用服务器与负载均衡器的配合。负载均衡器周期性地访问一个“健康性检查”的页面，网络服务器通过把这个页面设置为一个出错页面（HTTP返回码“503 Not Available”）或者一个带有出错信息的HTML页面，来通知负载均衡器它处于忙的状态。这样负载均衡器就知道不给那个服务器发送额外的工作了。当然，这只是对于网络服务有用，而且在所有的网络服务器都很忙、无法处理另外一个页面请求的时候，这个方法仍然会失效。

在基于服务的架构下，服务器可以提供一个健康性检查的查询给客户使用。这样，客户在发送请求之前会检查服务器的健康性。这种方法提供了很好的握手机制，带来的开销是服务器必须处理的连接和请求的数量都翻倍了。不利的是，一般的网络服务调用所花的时间只是建立和撤销TCP连接。在实际调用服务之前先进行健康性检查会使连接的开销翻倍。

握手在处理能力不匹配而导致慢速响应的时候比较有价值。当服务器发现它不能满足SLA的时候，它应该有一些方法来要求调用者回退。如果服务器是部署在负载均衡器的后面，它可以选择对于负载均衡器关闭响应，这样后者会把不响应的服务器清理出服务器池。当然这是一个比较粗鲁的机制。那么，你最好在自己实现的一些协议里面构建握手机制。

如果你调用的服务没有提供握手，调用断路器是一个权宜之计。在这种情况下，不是有礼貌地问服务器是否可以处理一个请求，而是通过断路器直接调用它，然后看它是否工作。

总而言之，握手是一个仍未被充分利用的技术，如果在应用层协议使用，会有很大的好处。它是防止裂缝在不同层次间传递的有效方法，比如前面提到的连锁失效的案例。

请记住

- 建立合作的需求控制机制

客户端和服务器的握手可以实现需求的限制机制，让需求可以维持在一定的可服务水平上面。客户端和服务器都需要实现握手机制。很多常见的应用层的协议，比如HTTP、JRMP、IIOP以及DCOM，都不进行握手。

① 参见 http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

② 许多客户端甚至将其他的200系列代码视为错误！

- 考虑健康性检查

如果协议中没有握手机制，那么实现应用层的健康性检查是一个补救的方法。当添加一个这样的调用的成本远远小于调用然后失败的成本，就可以尝试使用它。

- 在底层协议中建立握手机制

如果你要实现自己的基于套接字的协议，请在里面实现握手，这样的话如果端点没有准备好工作的时候，就可以通知对方。

5.7 测试装置



正如在前面的章节中看到的那样，分布式系统的失效模式很难在开发或者测试环境中被触发。为了更加完整地测试各个部件，我们经常依赖一个“集成测试”环境。在这个环境中，我们的系统和所有与它必须交互的其他系统集成在一起。

集成测试也有它的问题。我们需要测试什么版本？为了得到最大的保证，我们往往会测试在发布系统时我们所依赖的最新版本。我可以利用集合理论来证明，这种方法会限制整个公司一次只测试一个新的软件，但我把这个证明留给读者。而且，目前的系统互相依赖，构成各系统间互锁的网状结构，集成测试会变得很单一，一个全局的基础测试环境掩盖了整个公司真正运行的系统。这样的环境需要的变更控制比实际的生产环境更加严格。

这是一个更加抽象的难题（你也许会问：比集合论还要抽象？）。集成测试环境只可以验证当系统依赖的部分都工作正常时它是否工作正常。当然可能会触发远端的系统返回一些错误，但它仍然或多或少在按照规范工作。如果规范说“如果收到的请求中没有包含上次电话卫生清理的日期，那么系统就返回错误号 14916”，那么调用者可以强制产生这样的错误。不管如何，远端的系统仍然在按照规范工作。

而这本书考虑的是，所有的系统最终都可能不按规范工作。所以，远端系统不正常的情况下，测试本地系统的行为是非常重要的。除非远端系统的设计者已经包含了对于实际生产环境可能发生的全部错误模拟，不然，就会有在集成测试中无法验证的行为。

一个集成测试的好方法是让你测试大部分甚至全部这些失效模式。它要能够保留或者增强系统间的隔离，来避免版本锁定的问题，同时可以在很多地点进行测试，而不是仅仅在我前面所提到的单一的企业集成测试环境。

因此，你可以构造测试装置来模拟远端系统在每个集成点另外一端的表现。硬件和机械工程师已经使用测试装置很长时间了。而软件工程师虽然开始使用它，但用得还不够。一个好的测试装置需要足够狡猾，它需要和实际的系统一样讨厌和满怀恶意。测试装置需要在被测试的系统中留下伤疤，它要使被测试系统变得愤世嫉俗。

考虑构造一个可以替代远端 Web 服务调用的测试装置，因为远端调用使用网络，套接字连接可能会碰到如下的失败。

- 它可能会被拒绝。
- 它会在监听队列里面等待，直到超时。
- 远端回送一个 SYN/ACK，然后再也不返回任何数据。
- 远端只发送 RESET 包。
- 远端报告一个完整的接受窗口，但从来不消耗数据。
- 连接建立，远端从来不发送数据。
- 连接建立，包丢失重新传递导致延时。
- 连接建立，远端从来不回应收到数据，导致不断重新发送。
- 服务接收请求，回送一个响应头，然后再也不发送响应体。
- 服务每三十秒发送一个字节的响应。
- 服务回送了一个 HTML 响应而非期待的 XML 格式。
- 服务发送了上兆字节的数据，而我们只希望收到数千字节。
- 服务拒绝所有的鉴权凭据。



小乔爱问……

为什么不使有Mock对象

Mock对象是在单元测试中经常使用的技术。[①] Mock对象提供了一种可以被单元测试控制的、对于对象的不同实现，被测试的对象会使用它。例如，假设一个应用使用了一个DataGateway对象作为整个数据持久层的代理。实际的DataGateway需要处理连接参数、数据库服务器，以及一些测试数据。对于一个测试来说，耦合很多，由于隐藏的依赖，通常会造成测试结果无法重现。一个Mock对象去除了所有的外部连接，增加了单元测试的封闭性。Mock对象经常应用在软件层次的边界上面。

某些Mock对象可以在调用它们的方法时，抛出一些异常。这允许单元测试可以模拟某些失效的状况，特别是可以对应到异常的那些失效（假设真正实现的代码里面也会产生异常）。

测试工具和Mock对象不同，模拟对象只可以产生和已定义接口一致的行为。测试工具运行在单独的服务器上面，所以它并没有责任和某个接口保持一致。它可以触发网络错误、协议错误或者应用错误。如果可以保证所有的底层错误都可以被识别、捕获，而且抛出为正确类型的异常，那么就没有必要使用测试装置了。

① 参见 http://www.junit.org。

这些错误分为不同的类型：网络传输问题，网络协议问题，应用层协议问题，应用逻辑问题。稍微思考一下，你可以发现 OSI 七层模型中每一层的错误模式。在应用中增加各种条件分支或标志来模拟所有这些可能出现的错误是非常昂贵和奇怪的。谁会愿意在系统进入运营后，还去冒不小心开启一个模拟的错误的风险呢。集成测试比较善于捕获第七层即应用层的错误，但也不是全部。

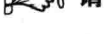
让你的测试装置像黑客那样行动。

而测试装置知道它的目的就是用来测试的，它没有其他的任务了。实际的应用不会直接调用底层的网络编程接口，测试装置可以。这样，它可以很快或者很慢地发送数据。它可以设置很深的监听队列，可以绑定到一个套接字却从来不响应连接请求。测试装置可以像一个黑客那样，尝试利用各种方法去破坏调用者。

对于各种不同的应用和协议，坏行为都是相同的，比如拒绝连接、连接缓慢、接收请求而不响应都可以应用到基于套接字的协议上面：HTTP，RMI 和 RPC。对这些协议，一个测试装置可以模拟很多种不同的坏的网络行为。我比较喜欢使用的一个技巧是使用不同的端口号来代表不同的异常行为：在端口 10200，它可以接受连接但从来不回应；端口 10201 接受连接，但从 /dev/random 里面拷贝数据回应；端口 10202 打开连接，然后马上关闭；诸如此类。这样的话，我都不需要改变测试装置的模式，一个测试装置就可以攻破很多应用。如果让多个开发人员在他们的工作机上运行测试装置，它甚至可以帮助进行功能测试。（当然，让开发人员运行他们自己的杀手级测试装置也是很值得的。）

请记住，测试装置非常善于攻破甚至是杀死你的应用程序。所以，测试装置要能够记录请求的内容，以免在应用程序死亡的时候，没有任何线索知道是什么东西杀了它。

测试装置可以设计得类似应用服务器，它可以有和实际应用测试相关的可以插拔的行为。一个单一的测试框架根据需要可以被实例化成不同的应用层的协议，或者是对应用层协议进行改变。



请记住

- 模拟规范之外的失效

调用实际的应用只可以让你测试那些应用特别可以产生的错误。一个好的测试装置应该可以模拟所有杂乱的实际失效模式。

- 给调用者加压

测试装置可以产生缓慢的回应、不产生回应或者产生垃圾回应。然后你看你的应用如何反应。

- 对于常见的失效使用共享的测试装置

不需要为每一个集成点设计一个独立的测试装置。一个“杀手级”的服务器可以监听很多端口，然后根据所连接的端口产生不同的失效模式。

- 补充而不是替代其他测试方法

测试装置是用来补充其他测试方法的，它不会替代单元测试、验收测试、FIT 测试等。所有的这些测试都是来帮助验证功能行为的。测试装置帮助验证“非功能性”的行为，同时维护了和远端系统的隔离。

5.8 去耦合中间件



中间件这个不太像样的名字，是指那些位处杂乱空间的工具，它们力图把从来就没有打算一起工作的系统整合起来。2000 年早期，中间件被重新打上“企业应用集成”的标签，一时成为一个热点，然后就又黯然回退到阴暗模糊、不受人注意的现实中。中间件占据了企业系统的重要的连接处，它是把企业中各个自动化的孤岛连接起来的桥梁。

中间件通常被描述为“管道工程”，这也往往隐含着它固有的杂乱肮脏，因为它要和不同的流程、不同的技术，甚至是对于同一逻辑概念的不同定义打交道。它的这种不那么酷的特性也能部分解释为什么基于服务的架构现在开始赢得人们的关注，取代了不那么迷人但更需要的中间件。

如果做得好，中间件可以同时实现系统的整合，又降低耦合。它通过在系统间传递数据和事件来整合，又通过让参与的系统不必了解其他系统的内部知识、不必调用其他系统来实现解耦合。因为集成点是引起不稳定的首要原因，这看起来是个好事情。

任何同步的调用/响应或者请求/回应的方法都强制要求调用者停止当前的处理等待。在这种模型下面，调用和接收系统都必须同时活动，它们在时间上是同步的，尽管可能在不同的地方。这个类别包含了远程过程调用（RPC）、HTTP、XML-RPC、RMI、CORBA、DCOM，或者任何和本地调用类似的方法。紧耦合的中间件会放大对于系统的冲击。同步调用是引起级联失效的危险的放大器。

不那么紧密耦合的中间件允许调用和接收方在不同的地点和时间处理消息。有名的 IBM MQSeries 以及任何基于发布/订阅的消息系统，都归到了这个类别，这个类别还包括通过 SMTP 或者 SMS 的系统到系统的消息传递。（后面两个协议经常包含了用无线技术实现的、而不是计算机软件中的消息代理。延时一般都比较高。）

图 5-4 描绘了各种中间件技术针对的耦合谱系。

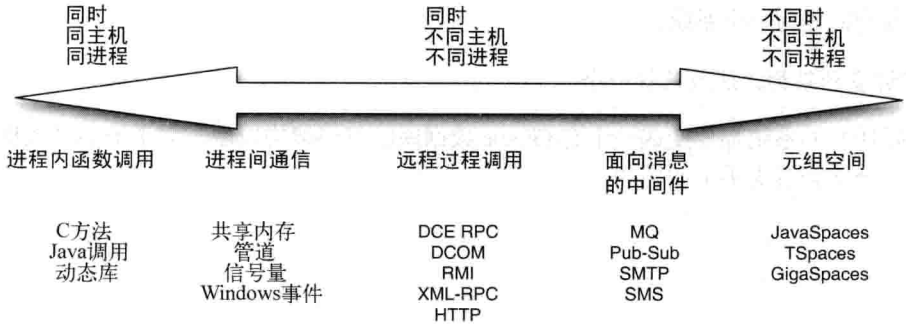


图 5-4 中间件的耦合谱系

基于消息的中间件把端点在空间和时间上面解耦合。因为请求系统不需要坐在那里等待回应，这种类型的中间件不会产生级联失效。

同步（紧耦合）中间件的好处是它的逻辑比较简单。假设客户请求授权来进行信用卡购物，如果这个请求是用远程过程调用或者 XML-RPC 来实现的，应用程序可以很清楚地决定是进行到下一步的付款流程还是回退到付款方法的页面。与之相比，如果系统只是发送一个消息，请求信用卡授权，而不等待回应，那么它一定需要决定如果这个授权请求最终失败，或者更糟糕的是没有得到回应，应该如何处理。设计异步的流程更难。这个过程必须可以处理异常队列、迟到的回应、回调（计算机到计算机以及人到人），以及假设。有时甚至需要项目出资人来决定，比如可以接受的财务风险是什么样的。

你可以应用本章里面的大部分模式，同时不会显著增加系统的花费。使用什么中间件的决定就不是这样了。首先，中间件产品很贵；第二，不同风格的中间件会带来不同的设计。你思维方法的变化带来的花费会很高。从请求/回应 RPC 模型切换到元组空间的模型，需要在不同层面上有不同的思维方法。

最后，中间件一般会在企业的层面去购买，因此，外部已经在你开始进行项目的架构设计之前，已经为你决定好了。

请记住

- 早作负责任的决定

其他稳定性的模式不需要对于系统的设计和架构进行大规模的改变。去耦中间件是架构设计，它渗入到系统的每个部分。这是那些需要尽早做出的无法逆转的决定。

- 通过完全解耦合来避免大部分失效模式

你越能够把单独的服务器、软件层次以及应用解耦合，那么遇到和集成点、级联失效、缓慢回应、被阻塞的线程等相关的问题就会越少。你也会发现解耦合的应用适应性更高，因为你可以

单独修改其中的任何一个系统。

- 了解多种架构，然后选择一个

并不是所有的系统都要变成一个有 Oracle 数据库的三层架构应用。多了解不同风格的架构，然后选择一个适合解决手上问题的。

第 6 章
稳定性总结

随着时间的推移，那些令人震惊的不可能同时发生的情况最终也会发生。如果你觉得“这样的事发生几率太低了”，或者有类似的吐槽的话，请想想看：三年时间，每天页面浏览量达上百万次，假设每页 50 条数据，你的系统有 547 500 000 000 次机会出错。这可是 5000 亿次机会啊！最近的天文观测表明，银河系有 4000 亿颗恒星，而它每天都在发生不可能的事情。

> 宇宙中每天都在发生不可能的事情。

故障不可避免。我们的系统，和我们依赖的系统，都会以大大小小的方式出现故障。稳定性反模式放大了坏事件的效应，它们让系统加速崩溃。避免反模式也不会阻止坏情况发生，但是它会帮你减少损失。

明智地运用稳定性模式能让软件运行更久，并经受住各种考验。成功运用这些模式的关键是判断。检查软件的需求，带着质疑和猜测查看其他的企业系统——姑且认为它们都可能会威胁你的系统。找出危险，并对危险运用适当的稳定性模式。多疑是良好的思维方式。

能经受得住就赢得了战役的大半。想想你的系统的成功概率。大约一半的系统在项目开发时就取消了，剩下的又有一半延期、超出预算，或者不符合需求。只有不到四分之一的项目投入到了生产环境，而其中大多数还会由于停机招致大量的损失，收入减少，维护成本高昂。

沉舟侧畔，你已经成功投入到生产中，值得骄傲！

可悲的是，问题不出现通常也不会被注意到。挽救一个拙劣的实现，你可以有机会像个英雄一样。可是，如果你一开始就设计了一个稳定的系统，就不会有人注意到你的系统不需要停机时间。事实就是这样。你交付了一个牢不可破的系统，用户也还会去抱怨些别的问题。用户天生就是这么当的。

实际上，对一个从来不崩溃的系统，用户最可能抱怨它响应慢。接下来，你将看看容量和性能问题，以及如何充分利用你的资源。

Part 2

第二部分

容量

本部分内容

- 第 7 章　案例研究：被客户压迫
- 第 8 章　容量概述
- 第 9 章　容量反模式
- 第 10 章　容量模式

第 7 章
案例研究：被客户压迫

7.1 发布倒计时

经过年复一年的努力，终于迎来了项目发布的这一天。我是在九个月前加入这个庞大团队的，团队有 300 多人，我的任务是帮助建立一个完整的系统，包含了网上商店、内容管理、客户服务和订单处理。这是公司未来七年的骨干系统，在我加入团队时，它已经启动了一年多。在开始的九个月里，我一直在加班：午餐就在办公桌旁解决，每天工作到深夜。即使天气很好的那几天，明尼苏达州的冬天也在考验着你的心理承受力。天亮得晚，黑得早，我们披星戴月地好几个月看不到太阳。这常常让人觉得是奥威尔式的噩梦，逃也逃不掉。我们不知不觉中就过了春天，这唯一适合在这里生活的季节。我在一个冬夜睡去，等我再醒来环视四周，发现夏天已经到来。

九个月后，我依然只能算是个新人。一些开发组成员已经这样度过了一年多。他们每天吃着客户捎回来的午餐和晚餐。即使在今天，有的人想起土耳其玉米饼还会瑟瑟发抖。

但是，今天是胜利的日子。所有的辛劳、沮丧，被遗忘的朋友、不愉快的离婚，都在我们发布之后烟消云散了。

营销组的成员聚集到会议室里召开隆重的启动仪式，喝着香槟，其中很多人自两年前的需求收集会议后就未再见过面。这些将含糊的梦想变成现实的技术专家们，围拢在布满显示屏的墙边，观看网站的顺利运行。

上午 9 点，项目经理按下大红的按钮。（确实有个大的红按钮，连接到隔壁房间的显示器上，那里有位技术人员刷新浏览器，投影到大屏幕上。）新的网站就像魔术一样出现在会议室的大屏幕上，我们在楼层的另一面，听到营销组在热烈欢呼，香槟酒软木塞迸开。新的网站诞生并运转起来了。

当然，真正的改变是从 CDN[①]（内容分发网络）开始的。对元数据的更新计划在中部时间上午 9 点，这些更新会在 CDN 网络服务器中逐渐扩散，大约经过 8 分钟就能在全世界范围生效了。我们期待在上午 9:05 时能看到新服务器上流量达到峰值。（会议室的浏览器配置为绕过 CDN 直接登录网站，也就是直达“原始服务器”。会要手腕迷惑人的不只有市场营销的人。）实际上，我们能够立即看到网站的流量。

CDN：内容分发网络，又称边缘网络。它是加速器，将图像和静态内容缓存在离用户最近的服务节点，这样可以消除多达 80%的来自 Web 服务器的请求。

到上午 9:05，服务器上已经有了 1 万个活跃会话。

9:10，网站上还有 5 万多个活跃会话。

9:30，网站上有 25 万个活跃会话。然后，网站崩溃了。

7.2 瞄准 QA

要想知道网站为什么崩溃得如此之快，我们需要回顾一下三年前。

现在很少能见到这样的新建项目了，原因很多。首先，没有这样的网站项目了。每个网站都是带有 HTML 前端的企业集成项目，大多数项目至少都有必须集成的后端系统。当后端和前端一起开发时，你可以想象，这将是更干净、优越、紧密的集成。这很可能会发生，但并不是自动实现，它取决于 Conway 法则。更常见的结果是，双方的集成目标是不断变化的。

> 每个网站项目都是企业集成项目。

替换整个商务系统也会带来重大的技术风险。如果系统不是建立在稳定的模式上，那么它很可能会遵循典型的紧耦合架构，这样系统的故障几率是其组件故障的概率之和。

即便系统是建立在稳定模式下的（案例中的不是），一个全新的系统意味着没人知道它在生产环境会如何运行。容量、稳定性、控制和适应能力都要打上巨大的问号。

早先我就意识到，开发团队是为通过测试而构建系统的，并非针对产品运行环境。在所有 15 个应用程序和 500 多个集成点中，每个配置文件都是针对集成测试环境构造的。主机名、端口号、数据库密码等分散在上千个配置文件中。更糟糕的是，程序的一些组件是按 QA 的拓扑结构

① 事实上，在周一的网站启动之前，CDN 在周六时已经让世界对网站先睹为快了。但是不知何故，元数据的更改完全不正确，而且源服务器切换发生在周六下午。那时，网站是可见的（而且能接受订单），但是只有部分内容被加载了，然后我们花了大约一个小时确定问题源于 CDN。我们又花了一个小时恢复更改，并扩散到整个 CDN 网络。哎呀！

设计的，而这并不符合产品运行环境，例如产品运行环境中会有防火墙，在 QA 中却没有。（这是一个常见的“捡了芝麻，丢了西瓜”的决策，虽然节省了几千美元的网络设备，但是在停机和部署时损失更大。）此外，在 QA 环境，某些应用程序只有一个实例，但是在产品运行时会有多个集群实例。在很多方面，测试环境反应的系统架构是过时的，每个人都知道这在产品运行时就完全不同了。但是，这很难在测试环境中改变，大多数开发团队都选择忽视差异，而非花上一两星期去做日常的部署测试。

Conway 法则

在 1968 年的一篇关于数据处理的文章中，Melvin Conway 描述了一个社会现象：“任何一个产品设计都会不可避免地反映出设计它的团队组织的沟通结构。”有时通俗地表述是：“由 4 个团队来开发一个编译器，那么最终的编译器也很可能包含 4 个编译过程。”

虽然这听起来像是个呆伯特漫画，但它实际上源自对软件设计进行的严肃而又中肯的分析。Conway 认为，对于系统内或系统间的接口，两个人必须以某种方式对接口规范进行沟通。没有沟通，就不能建立接口。

请注意，Conway 指的是组织的“沟通结构”，这并不等同于组织的形式结构。如果不同部门的两个开发者能直接沟通，那么沟通会在系统中直接对应为一个或多个接口。

我发现，Conway 法则在创建软件所包含的沟通结构时非常有用；而在描述结构上，这种软件结构映射能帮助理解组织的真实沟通结构。

Conway 的原始文章可以在作者的网站http://www.melconway.com/research/committees.html上找到。

考察产品配置时，我认为仅仅需要找出解决过这些问题的人就行了。我的问题是：“产品配置是检入到源代码控制仓库的什么地方的？”“谁能告诉我，产品中的什么属性需要覆盖？”

有时候，你问了这些问题，但没人回答你，也就是说没人知道答案。还有些时候，没人想回答这些问题。对于这个项目，两种情况都有。

我决定整理一个属性表，包括了产品可能会改变的属性：主机名、端口号、网址、数据库连接参数、日志文件的位置，等等。然后，我找开发人员要答案。一个叫作“主机”的属性是含糊的，特别当测试时主机上运行有 5 个应用程序。它可能是指“我自己的主机名”，也可能指“允许调用我的主机”，再或者是“我想来洗钱的主机”。在弄清它在产品中应该指什么之前，我必须知道它过去指的是什么。

一旦有了关于产品中需要修改属性的整体构想，就可以定义产品的部署结构了。上千个文件需要在产品运行环境中改变。它们需要在每个软件发布版本中重写。在三更半夜对每个发布版本

都手动修改这上千个文件，绝不可取。此外，有些属性重复出现了很多次。比如，只是修改数据库的密码就需要对二十多个服务器修改上百个文件，如果网站规模扩大，问题会更糟。

面对这个棘手的问题，我做了任何优秀开发者都会做的事：添加一个间接层。关键是创建一个覆盖的结构，与应用程序代码库保持隔离。这种覆盖结构要保证不同环境的每个属性都放在一个地方。然后，每个新的版本都可在不需要重写产品配置的情况下部署。这样的覆盖结构也有利于使产品数据库的密码不同于测试环境（开发人员能够访问），也不同于源代码管理系统（公司的任何人都能访问），因此能够保护客户的隐私。

在搭建产品环境时，我在不经意间主动协助了负载测试。

7.3 负载测试

对于一个未经试验的系统，客户都知道负载测试至关重要。客户安排了整整一个月来做负载测试，我此前从来没有见过这么长时间做测试的。在网站发布之前，营销部门已经声明它必须要能支持 25 000 个并发用户。

负载测试通常是自动的过程，无需人工干预。先制订一个测试计划，创建出测试脚本或让供应商创建脚本，配置负载产生器和测试分发器，然后在夜里启动测试。第二天，测试完成后，你就能分析收集到的数据了。分析结果，修改代码或配置，然后再安排下一次测试。在下次测试之前，时间已经过去了三四天。

我们知道需要加快测试周期。因此，我们召集了一些人开电话会议：测试经理，负载测试服务的工程师，开发团队的架构师，查看数据库使用情况的 DBA，和我（监视和分析应用程序和服务器）。

负载测试既是艺术也是科学。由于不可能复制真实生产环境的网络流，因此只能通过使用流量分析、经验和直觉来尽可能地模拟现实。流量分析只会告诉你变化：浏览模式、每个会话的页数、转换率、思考时间分布、连接速度、目录访问模式等。得凭着经验和直觉为不同变量分配重要性。思考时间、转换率、会话持续时间和目录访问是最重要的考虑因素。我们的第一个脚本混合了“浏览者”、“搜索者”和“买家”。90%以上的脚本会访问主页和某个产品的详情页。这些代表了那些每天访问网站、寻求打折信息的老手。我们乐观地估计有 4%的虚拟用户会走完购买的全程。这个网站与大多数电子商务网站一样，结账是最费事的。它涉及诸多外部集成（CCVS，地址规范化，库存检查，可购买物品检查），几乎比其他任何会话都需要更多的页面。正常结账的用户通常要访问 12 个页面，而只是浏览下网站就走的用户最多只访问 7 个页面。我们相信混杂的虚拟用户比起现实来说对系统要求更苛刻。

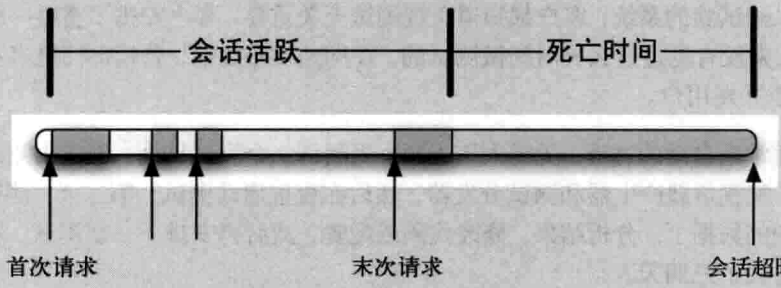
在第一次测试运行时，测试顶峰达到 1200 个并发用户，当时网站就彻底挂了。每个应用服务器不得不重启。我们大约需要扩容 20 倍。

什么叫并发用户？

负载测试公司经常谈论“并发用户”（concurrent users），他们实际上指的是“机器人”（bots）。有些项目赞助商用这个词表达“会话”（sessions）。其实没有所谓的“并发用户”。除非你是在构建一个两层的客户/服务器系统，用户能直接与数据库相连，否则并发用户就是虚构的。

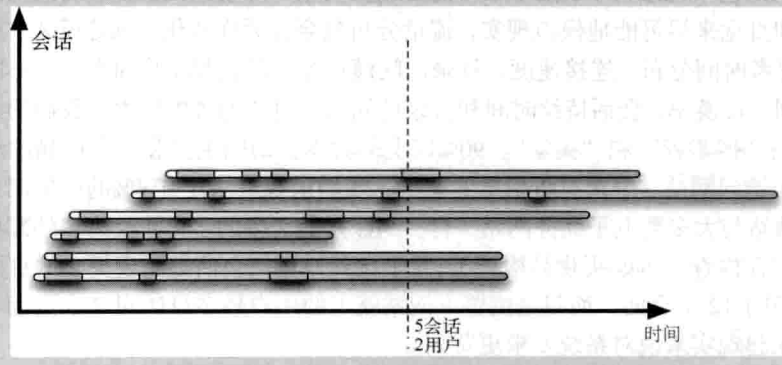
统计并发用户来判断系统容量是个具有误导性的方法。如果 100%的用户都看了首页后就离开了，容量会表现得很高，比 100%的用户都购物高得多。

你无法衡量并发用户。没有长时间的连接，只是些离散的脉冲。服务器接收到由某个标识符联系起来的一系列请求，这些请求组成一个会话，这样的抽象是为了使程序编写更容易。



请注意，用户在死亡时间开始时就流失了。服务器不能区分不再继续点击的用户和未曾点击过的用户，因此使用了超时机制。自用户最后一次点击后的数分钟内会话保持存活。

这意味着会话比用户持续更久。对会话计数会超过用户的真实数字。



看看这些活跃的会话，其中一些注定会过期。活跃会话的数字是网络系统最重要的量度，但是它不该和用户计数相混淆。

在接下来的三个月，我们每天开 12 个小时的电话会议，进行了各种有意思的探索。一个难忘的夜晚，负载测试厂商的工程师看到测试农场的所有的 Windows 机器开始下载并安装某些软件。这些机器是在我们开会时被黑客入侵了！还有一次，我们似乎到达了带宽的上限。果然，某个 AT&T 工程师注意到有个子网使用了“太多的”带宽，因此他封锁了产生 80%负载的链路。但是，除了这些插曲之外，我们的网站还是取得了比较大的改进。每天都发现了新的瓶颈和容量限制，我们能在一天内修改配置。代码更改要花更长的时间，但是也能在两三天内修改好。

我们甚至在一星期内就完成了一些重要的架构改变。我会在下一章讨论这些改进。

之前的网站运营回顾也让我们有机会创建脚本、工具和报告重要的问题。

经过三个月的测试和 60 多个新的程序版本，我们将网站的容量提高了 10 倍。它已经能处理 12 000 个活跃的会话，我们估计这代表同时有 10 000 个客户（前面说过要谨慎进行客户计数）。此外，当压力超过 12 000 个会话时，网站也不再崩溃了，虽然会变得有点“容易断线”。在这三个月，市场部门重新评估了发布的目标。他们决定网站慢也比没有好。他们也不再要求 25 000 个并发用户，认为 12 000 个会话就足够了。每个人都期望我们在旺季来临前能做出重大的改进。

7.4 被大量会话所杀

在负载测试都完成后，发布当天却是怎么了？网站为何如此糟糕地崩溃了？我们首先想到的是市场部门极大地低估了需求，很可能广大客户已经对新网站充满了期待。但这个想法很快就被打消了，因为根本就没有告诉客户网站的发布日期。或许是一些配置错误？或者生产与测试环境还不匹配？

查看会话的个数一下子让我们找到了问题的原因。正是会话的数量干掉了网站。会话是每个应用服务器的阿喀琉斯之踵。每个会话都要消耗资源，尤其是内存。启用会话复制后，发生页面请求时每个会话都被序列化然后传到会话备份服务器。这意味着这些会话都要消耗内存、CPU 和网络带宽。所有的会话都从何而来？

会话是每个应用服务器的阿喀琉斯之踵。

最后，我们意识到噪音是最大的问题。我们所有的负载测试都是通过脚本来模拟使用真实浏览器的用户来完成的。他们从一个页面跳到下一个链接的页面，这些脚本都是用 cookie 来跟踪会话。它们对系统很温和，但是真实世界的用户却是十分粗鲁、原始和邪恶的。

产品环境中，总是能发生你没法预测的坏事。我们面临的困难之一是搜索引擎。搜索引擎为网站带来 40%的访问量。不幸的是，在切换当天，它们将客户推到了旧的 URL 上。网络服务器配置为将所有的 html 请求发送给应用服务器，因为应用服务器能够跟踪和报告会话。这意味着

来自搜索引擎的每一个用户都会在服务器上创建一个会话，而只是导致了 404 错误。

还有一个大问题就是搜索引擎抓取网站。有些爬虫（特别是些不知名的搜索引擎）并不跟踪 cookies。它们不想影响营销数据或广告收入。这些爬虫通常希望网站通过 URL 重写来支持会话跟踪。但是，没有了 cookie，它们会为每个页面请求创建一个新的会话。会话会驻留在内存中直到 30 分钟后过期。我们发现有个搜索引擎每秒创建了 10 个会话。

接下来是抓屏器（scrapers）和购物机器人（shopbots）。我们发现了大约十个高容量抓屏器。它们有些很聪明，能够避免暴露来源。来自不同子网的页面请求掩盖着它们其实出自同一个源头。事实上，相同 IP 地址的连续请求会使用不同的用户代理字符串来掩盖其来源。

用户代理：由浏览器发送的标识自己的 HTTP 首部。包括微软的 IE 浏览器在内，几乎所有浏览器都是某种形式的 Mozilla。

但是，ARIN[①]还是能识别出属于同一个实体的源 IP 地址。这些商业的抓屏器实际上是为了销售订阅服务，想了解竞争对手商品价格的零售商会订阅这些报告。它按周或按日来报告竞争对手的商品和价格。这是有些网站不显示销售价格的原因，除非你将货物放进了购物车。当然，这些抓屏器都没有妥善地处理 cookies，因此它们都在制造新的会话。

我们还需要处理业余的购物机器人。有些源 IP 地址会不断地访问旧网站的某个产品的详情页面，几乎每分钟一次。这需要我们花时间去找出这个产品，结果是 PlayStation 2。三年前 PS2 曾出现一次大短缺，三年后，当年那个查找存货数量的脚本依然在运行，它当然也会创建更多的会话。

最后，还有些源，我们称为“随机的怪玩意儿”。例如，我们发现某海军基地的一台计算机会在常规浏览会话中出现，最后一次页面请求的 15 分钟后又会出现。我们看到它反复请求最后的那个 URL，这自然又创建了更多的会话。

7.5 测试的鸿沟

尽管进行了大量的负载测试，在面对真实世界时它依然崩溃了。我们的测试中遗漏了两点。

首先，我们按照理想中的方式测试系统。测试脚本会请求一个网址，等待响应，然后请求另一个。测试脚本没有尝试每秒 100 次进入相同的网址，而且不用 cookies。如果它们这么尝试，我们很可能会说这测试“不实际”，并且会忽略服务器的崩溃。由于网站只适用 cookies 来进行会话跟踪，没有使用 URL 重写，我们所有的负载测试脚本都使用了 cookies。

总之，所有的测试脚本都遵守规则。这就好比一个程序测试只按正确的顺序点击按钮。我所知道的大多数测试人员都够不守规矩的，如果你告诉他们使用应用程序的“愉快路径”，他们绝不

① 参见 http://www.arin.net。

会这么听话。负载测试也应该如此。“噪音”也许只会损害你的部分容量，可是它能让网站崩溃。

> 不要只遵循“愉快路径”。

其次，应用程序开发人员没有引入切断坏事的安全机制。出现问题时，应用程序还会让线程继续进入危险区域。就如同在迷雾茫茫的高速公路上出现了车祸，新来的请求线程只能不断拥积下去。

7.6 后果

网站发布后日复一日的辛苦努力终于带来了成效。CDN 的工程师们在发布前修改了发现的错误。一天之内，他们使用高超的脚本使网站避免被恶意攻击。他们添加了一个网关页面来完成三个重要功能。首先，如果请求者未妥善地处理 cookie，这个页面会将浏览器重定向到一个独立的页面，在那里解释了该如何启用 cookie。其次，我们可以设置一个阈值来决定允许的新会话比例。如果设置为 25%，那么只有 25%的请求会进入真正的主页，其余的请求会收到一个礼貌的消息，请他们过一会再进入网站。接下来的三个星期，我们安排一个工程师一直在密切地关注会话数量，一旦数量失控，随时降低阈值。如果服务器负载过高，就得花一个小时的时间才能恢复，所以使用这个阈值避免饱和是非常重要的。第三，我们可以禁用某些 IP 地址的访问。只要我们发现有购物机器人或大量请求，就会将其添加到阻塞列表中。

所有这些功能都应能作为应用程序的一部分来完成，但在发布后的紧张节奏中，从效率考虑让 CDN 帮我们完成更好。我们自己还有很多着急的事情要做。

主页已经是完全动态生成的了，从分类下拉菜单的 JavaScript，到商品的详情页面，甚至页面底部“使用条款”的链接。应用程序平台的主要卖点之一就是定制化。市场部门热衷于这个功能，但是没有确定如何使用它。因此，主页被生成并每天被访问 500 万次，每次都是一模一样的。需要 1000 多个数据库事务来生成这个页面（即便数据已经缓存到内存中，由于平台的工作方式，事务还是会产生）。这个 JavaScript 的下拉菜单的滚动特效需要浏览 80 多个目录。而且，流量分析表明每天主要的访问都只是集中在主页上。它们大多数都没有使用 cookie，所以定制化是不可能的。还有，如果处理主页时还需要应用服务器，这需要花费时间，还会在内存中产生一个保存 30 分钟的会话。因此，我们很快用脚本建立了一个主页的静态副本，用以服务那些没有 cookie 标识的客户。

你曾看过大多数商业网站上发布的法律声明吗？他们说了很多话比如“通过查看这个页面你已经同意了下面这些条件……”，结果是这些条件只出于一个理由：当零售商发现了抓屏器或者购物机器人，他们能找来律师对付他们。网站发布之初，我们也让法律部门的人忙着起草这些条款。我们发现了一些违法机器人抓取我们的内容和价格后，就通过律师发出警告，大多数时候，

这些机器人就会停手了。（不过就像将狗轰离饭桌一样，它们总是还会回来的，有时会换个伪装。）

这个混乱时期最传奇的一件事发生网站发布的那个星期。IT 运营经理找到 6 台符合我们配置的新服务器，它们被其他部门征用，但是还没有投入使用。经理把这几台服务器重新划拨给网站（当然又去订购了 6 台）。我们的一个系统管理员连续花了 36 小时准备这些服务器：安装操作系统，配置网络，配置文件系统，访问 SAN，一直看着。他一完成工作，我们就让人将他载回宾馆，估计他立刻就崩溃在房间了。然后我当天就安装了应用服务器的程序，并做好配置。我们在两天时间里让应用服务器的容量翻倍了。

应用服务器的会话失效备援机制是基于序列化的。用户会话绑定到原始服务器实例上，因此所有新的请求会返回到已经保存用户会话的实例上。每个页面请求后，用户会话就被序列化，然后发送到“会话备份服务器”。会话备份服务器在内存中保存会话。如果原来的实例崩溃了，后续请求会通过负载管理器定向到新的实例上去。新的实例然后会从会话备份服务器上加载用户的会话。这个机制工作良好，伸缩性也特别好，因为所有的会话都保存到内存中而不是数据库或硬盘上；也就是说，只要会话数据少，它的伸缩性就很好。例如，它通常包括用户 ID，购物车的 ID，还可能有当前搜索的一些信息，比如关键字和结果页索引。通常它不会将整个购物车都序列化，也不会序列化整个搜索结果（可能会高达 2000 个结果）。可悲的是，我们在会话中发现它就是真么干的，我们别无选择只能关闭会话失效备援。

所有这些快速响应操作都有一些共同的特征。首先，临时修复不临时。大多数这些操作会保留一两年。其次，它们都消耗了大量的钱，主要是失去了不少收入。很明显，被阈值阻隔的客户会离开而不会下订单（至少不会在这个网站下单）。关闭会话失效备援意味着结账中的客户在实例崩溃时都不能结账。例如，他们没法进入订单确认页，而是被退回到购物车页面。大多数结账中的客户就会直接离开网站了。将主页静态拷贝就难以完成定制化了，即便这是整个项目架构的原始目标。将应用服务器的硬件数量加倍直接增加了成本，而且也带来了运营的消耗。最后，还有来年必须从事补救工程而非新推创收功能的机会成本。

最糟糕的是这些损失都是不必要的。现在距网站启动已经两年多了。而今天，网站能处理 4 倍多的负载，却只要更少的服务器，硬件也没有更新。这一切是软件进行了改进。如果原来网站按现在的方式构建，工程师就能加入市场部门的聚会，畅饮香槟而不是忙着救火了。

第 8 章
容量概述

在网络泡沫之前，一则 IBM 的广告很好地诠释了系统部署的困境。一群焦急的企业家聚集在屏幕前观察新网站的启动。“收到的订单”数目逐渐升高，大家欢欣鼓舞。当计数器接近平稳的高值时，他们的欢呼转变成了不安的沉静。

我了解这种感受。这和前一章电子商务网站启动团队的感受一样，充满着沉重和无助。另一方面，我也见过一个系统在 18 个月里持续改进，处理了四倍的商业需求，却只使用了最初三分之二的硬件资源。这种容量上的进步完全来自软件设计的变化。

这一章展现了一种思考容量的方法，提出了影响容量的一些具体的模式和反模式。下一章会审视容量计划的商业方面。

8.1 定义容量

营销人员和经理们张口闭口就是性能和容量等词语，而架构师和开发者需要更精确的定义。我们先来定义一些术语。

性能（performance）衡量了系统处理单个业务的速度。它可以被隔离地度量，也可在负载时测量。系统的性能对吞吐量有重要的影响。客户虽然也常常使用性能这个词，但其实他们并不真正关心性能。客户关心的是吞吐量或容量。另一方面，最终用户也不关心整体容量，他们只关心自己提交的事务处理的性能。他们不可能登录到服务器来查看应用程序是否在运行，他们只知道，如果响应时间超过了预期，就表明系统崩溃了。

吞吐量（throughput）描述了系统在给定时间段所能处理的事务的数量。系统性能影响着吞吐量，但影响却不一定是线性的。吞吐量总是被系统瓶颈所限制，优化非瓶颈部分的性能并不会提高吞吐量。

可扩展性（scalability）常用在两个方面。首先，它能描述不同负载下吞吐量的变化。每秒请

求的响应时间图能表征可扩展性。其次，它也指系统支持的扩展模式。我会使用可扩展性来描述给系统增加的容量。

最后，在一定负载下对每个事务保持可接受的响应时间的系统最大吞吐量，称为容量（capacity）。

请注意，容量的定义包括了一些重要的变量。对于容量，不再是一个简单固定的数值。如果负载变化（比如假期来临，用户对各种服务兴趣大增），容量也可能完全不同。

这个定义还需要一个判断。“可接受的响应时间”究竟是多少？对于电子商务零售商，两秒以上的响应时间就可能让客户流失；对于金融交易，这个时间可能要更短，以毫秒计；而用旅行预订系统进行搜索，可能允许 500 毫秒的响应时间，预定的确认则能允许 30 秒。

8.2 约束

关于容量最棘手的问题是非线性效应。我们的大脑能够在潜意识里快速求解多个复杂的方程，从而完成开车或者抓住飞来的棒球等任务。但是，当考虑容量的时候，我们总期望它是个线性的问题。你是否问过这样的问题：“如果我们 50%的 CPU 占用可以处理 10 000 个用户，那么 CPU 满负荷时能处理 20 000 个用户对吗？”

对每个系统，只有一个约束条件真正决定了系统的容量[①]，就是那个首先达到极限的限制条件。一旦达到这个约束，系统的其他部分就要么按部就班工作，要么集体歇菜。例如，假设这个约束是数据库服务器。如果你使用 Oracle，并开启多线程服务器选项（MTS），那么服务器配置了多少个守护进程就能处理多少个并发请求。比如是个小数据库服务器，只有 50 个进程处理所有的请求，那么第 51 个请求就需要排队等待了。而提交这第 51 个请求的应用服务器本来有能力处理更多的页面，但它要等着数据返回。同样，网络服务器也空闲了，它要等待应用服务器的相应。

再假设应用服务器的内存成了约束条件。每个用户会话都需要消耗一些内存。一旦所有的可用内存消耗完，新的会话就会让应用服务器分页调度。同时，网络服务器还在等待应用服务器的响应。应用服务器遇上麻烦时，数据库服务器正好乐得能够休息一下了。

任何非约束指标对于提高容量都无意义。如果你的数据库服务器已经忙得不可开交了，提高网络服务器的 CPU 使用率也不能提高整个系统能处理的会话量。不过这也意味着一旦你发现了约束，就可以改善约束条件，并可靠地预测容量上的改进效果。如果内存是瓶颈，那么提高内存就能提高容量，当然，这时候可能还会出现其他的约束。

系统容量的理解需要系统思维，正如彼得·圣吉在《第五项修炼》[Sen90]中所描述的，系统

① 参见 *The Goal*[Gol04]中的约束理论。

思维是借助动态变量、时间的改变、相关的联系来思考的能力。没有一个简单的公式能推导出包罗万象的“容量指标”。开始从整体上思考系统，找到关键的驱动变量，比如“每秒页面请求”之类的。在整个系统的级别要找出这个驱动变量，一个简单的方法就是寻找那些你控制不了的东西，比如用户需求、时钟、日历等等。驱动变量常和后继变量相关，后继变量会伴随驱动变量改变。这些直接可测量的性能指数是：CPU 占用率，可用内存，I/O 速率，页交换速率，网络带宽等。负载测试、压力测试、产品系统的监视和数据分析能帮助确定和驱动变量相关的后继变量[①]。请注意，一个后继变量可能会和多个驱动变量密切相关。

从系统整体角度，你能先对系统分层或划分子系统，然后再查找驱动变量和后继变量。这些通常逐层改变。比如，数据库的 I/O 驱动着应用服务器的响应时间，而后者又影响着网络服务器的内存占用。同样，一个关系中的后继变量可能会是另一个关系中的驱动变量。这种错综复杂的相互关联表明了主驱动变量会如何改变整个系统。

系统的约束会成为某个后继变量的限制。当约束变量逼近其极限值，就不能再满足驱动变量的需求。在变量窗口的关联分析中，你会看到在约束变量达到极限时，约束变量和主驱动变量间存在明显的强联系。这时，关联崩塌，因为需求持续增长，但是服务却下降了。这就产生了负载测试图中的阿喀琉斯之踵。短板的快速崩塌标志着约束已经形成。

一旦识别了约束，剩下的就容易了。为了提高容量，必须增加能改善约束的资源，或者减少资源的使用。

8.3 关联

你经常能发现，某一系统层影响到了系统的另一层。例如，如果某一层的需求超过了容量，这一层的性能也会下降。它会响应缓慢甚至不响应。慢响应其实比不响应更糟糕。这时，这层的响应缓慢会给另一层带来连锁反应。这可能使我们难以区分究竟是容量问题还是稳定性问题。第 7 章里提到的电子商务网站在发布期间，整个团队遇到了一个严重的容量问题，直接导致了系统稳定性问题。

8.4 可扩展性

随着时间的推移，成功的系统往往也遇到更多的需求。某些时候，系统需要更多的容量，而这通常要求更多的硬件资源。一个水平扩展的系统能够通过增加服务器来升级容量。而垂直扩展的系统则需要对现有服务器升级。这些有时候也被称为“变宽”或“变大”。

处于同一同构资源池中的服务器，拥有一个负载均衡器或虚拟 IP 地址，都可以做水平扩展，

① 寻找高相关系数，介于 0.8（高度相关）和 1.0（完全相关）之间。

如图 8-1。当每台服务器只管自己运行而不用关心其他服务器时，你的系统就是完美水平扩展的。这些“无共享”架构在容量上几乎能做到线性增长。服务器数量翻倍也几乎就意味着容量加倍（除非增加的负载又促使新的服务被提交）。集群架构也允许横向扩展，虽然它们的效益还做不到线性增长，这是因为集群管理会带来一定的开销。

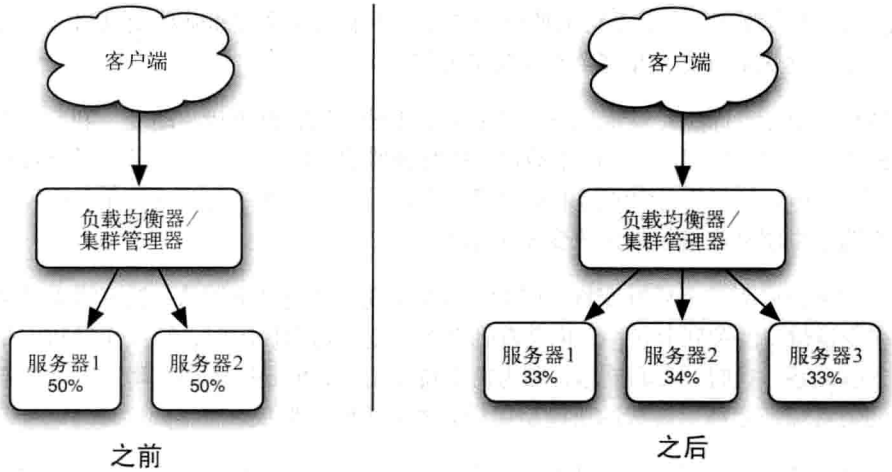


图 8-1 水平扩展

网络服务器是完美的水平伸缩型的，Ruby on Rails 服务器就是如此。诸如 WebSphere, WebLogic 和 JBoss 等 J2EE 应用服务器都是通过集群来实现水平扩展的。

有时候，平行地增加更多服务器并不实际也不可能。这时，每单个服务器要求越大越好。比如，数据库服务器，当你试图集群三个以上的冗余服务器时，它就变得很笨重。更好的方式是成对地运行并带有失效备援。

垂直伸缩需要服务器机箱有足够的空间来容纳更多的 CPU 和内存，如图 8-2 所示。一旦机箱达到最高极限，再向单个机器增加更多马力就需要做叉车式升级。能容纳许多 CPU 的机箱往往是一个产品线里最昂贵的，因此垂直可扩展架构比水平扩展型会有更高的初始成本。水平扩展架构使你的基础设施开销更加灵活，你不会让大量的资金束缚在一些大机器上。相反，你可以起初拥有尽量少的服务器，然后逐渐增加你的投资。（这还可以帮助你利用货币的时间价值，而不是迫使你一开始就耗资巨大）。

叉车式升级：换掉服务器的整个机箱。大的数据库服务器重达几百斤。

8.5 容量的神话

人类素来就相信一些奇怪的事情，比如中世纪巫婆会像鸭子一样漂浮，或者 CPU 使用率的

线性投影能告诉你系统的容量。有些迷信是无害的，如果运营经理认为只要他穿上夏威夷衬衫系统就会崩溃，你最好不要去打击他的迷信。但是，有一些错误的认识却会让公司损失数百万美元。

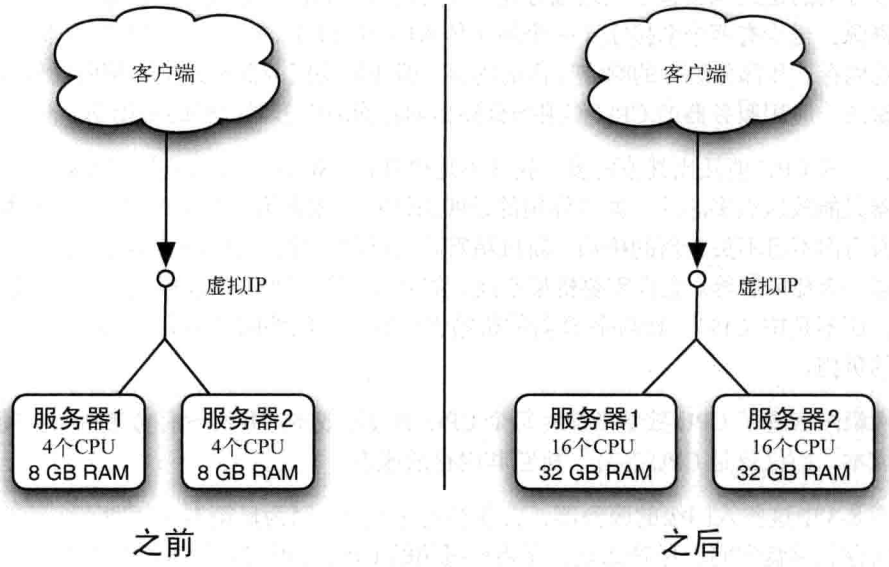


图 8-2 垂直扩展

CPU很便宜

我曾见识过不合格的开发者为他粗放的设计狡辩，他油腔滑调地说 CPU 很便宜。就好比说“花生酱很便宜，因此我要用三倍的花生酱来做这个三明治”。在 20 世纪 60 年代，一台电脑相当于一个程序员好几年的薪水。因此，程序员会花数小时甚至几天的时间去压榨出程序的每一个时钟周期。这正是使用廉价资源（程序员的劳动）充分利用昂贵资源（计算机）的案例。然而今天，一个 CPU 只相当于程序员半天时间的价值，为什么还要花时间去优化 CPU 使用率呢？

这种硅芯片本身可能很便宜，至少比过去便宜很多，但是 CPU 时钟周期并不便宜。每个 CPU 周期都要消耗时钟，带来延迟，浪费时间的应用程序会让用户等待较多的时间，而用户最不喜欢的就是等。对于网络系统，应用程序的延迟有双重效应。额外的处理会使应用服务器本身增加负担。假设应用程序让每个事务多花 250 毫秒，如果程序一天处理 100 万个事务，那么就需要多出 69.4 小时的计算时间。假设每台服务器的负载是 80%，就需要增设 4 个服务器来处理这个负载。

> 事实上，如果每个事务多花 250 毫秒，则 CPU 的计算时间每天多 69.4 小时。

由于大多数应用服务器采用线程池的方式处理事务，于是就存在吞吐量非线性效应。请求处理线程从池中签出的时间越长，传入请求必须加入队列等候而非立即执行的概率就越高。

应用服务器的延迟也损害了网络服务器。当网络服务器在等待应用服务器响应时，它会控制一些闲置资源：至少有两个套接字（一个用于传入的 HTTP 请求，一个用于应用服务器），用于请求状态的内存，和部分缓冲的响应要占的内存。即便网络服务器对连接不用做任何事，这些资源也是稀缺的。应用服务器的 CPU 使用率直接影响着网络服务器内存的使用率。

最后，一些 CPU 消耗比其他的多。我并不是说 PA-RISC 芯片会比英特尔 X86 芯片贵。我是指一台机器只能放这么多芯片。如果你用的是四路机箱，那么第 5 个 CPU 带来的成本就是不成比例的，因为你不得不买个新的机箱。新机箱需要自己的内存，可能还有本地磁盘、网卡、光纤通道适配器、冷却风扇等。它也需要机架空间，如果太大甚至要占用地板面积。你的软件可能按主机授权，而不是按 CPU。监测/管理新的机箱也有成本。额外的机箱会给数据中心的冷却系统带来额外的负担。

下面两幅图表现了 CPU 数量增加时每个 CPU 的边际成本。我们没有考虑机架空间、软件成本和管理成本，而仅仅是 CPU 芯片、机箱和内存的成本。

对于图 8-3 中这些入门级的服务器，成本差别并不大，因为机箱本身耗费不多。差别主要来自增加的内存和磁盘空间。尽管如此，需要加机箱的 CPU 比不需要加机箱的 CPU，成本要高出 20%。也就是说，第 3 个 CPU 的成本是第 2 个的 1.2 倍。

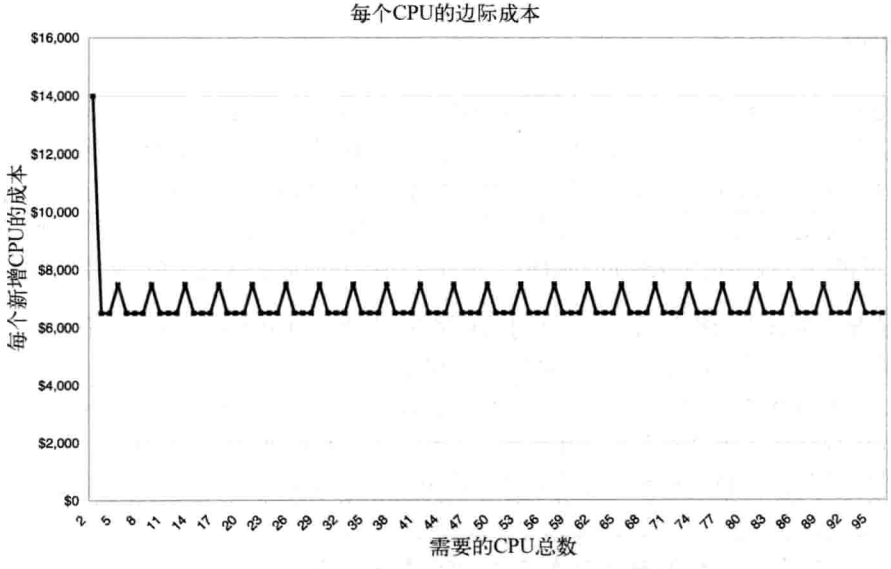


图 8-3　SUN 440 服务器中每个 CPU 的成本

对于高端的服务器，这张图就变化很大。高级的系统在扩展时需要更高的成本。图 8-4 中的 Sun Fire 6900 的机箱耗费超过 20 万美元。而最低配置的 Sun E25K 机箱的耗费甚至会超过 100 万美元。显然，增多 CPU 数量就会导致成本的峰值，E25K 能放 72 个处理器，但你需要付 100 万美元去为应用做更多的分析和优化。在投下 100 万美元之前，你必须确保没有浪费任何时钟周期了。你确信必需这第 73 个 CPU 吗？

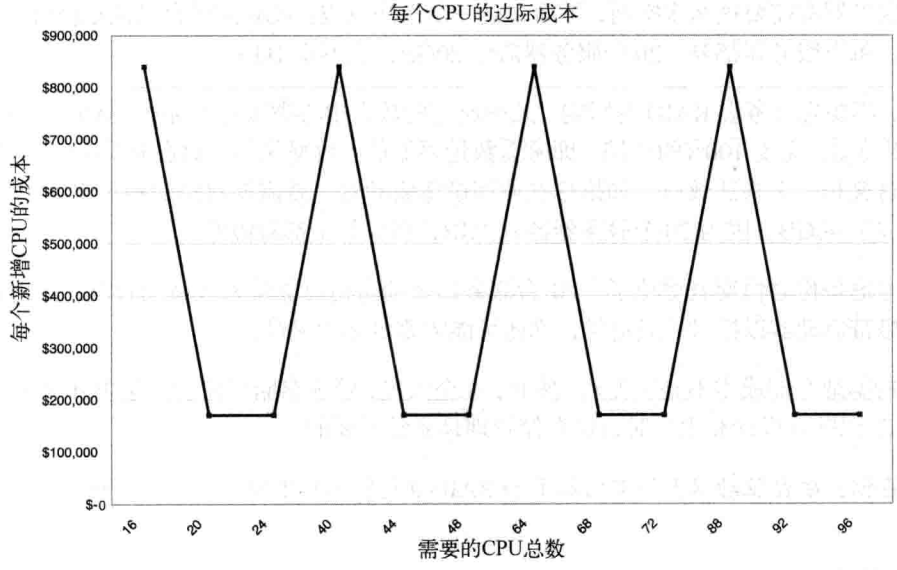


图 8-4 SUN FIRE 6900 中每个 CPU 的成本

存储很便宜

常听到有人说磁盘空间很便宜。谁说不是呢？毕竟，十年前存储的成本约为每兆字节 1 美元，现在每兆不到 50 美分[①]。确实，SCSI 硬盘并不便宜，但是驱动器足够便宜，你的笔记本里就能收藏上你拍摄过的所有照片。

麻烦的是，存储的现代定义不仅仅是个驱动器。存储是驱动器、互连、分配、冗余和用来备份高端服务的整个管理系统。存储更多是一种服务，而非今天大企业里的一块硬件商品。它开始于驱动器，但并不止于此。

存储是项服务，而不是设备。

磁盘可能位于本地服务器。这时，每个服务器都需要有足够的空间容纳以下内容：

① 来源：Pricewatch (http://www.pricewatch.com)，2007 年 1 月 13 日；500GB SATA 硬盘需要 147 美元。

- ❑ 操作系统
- ❑ 应用程序
- ❑ 本地配置或数据
- ❑ 日志文件
- ❑ 应用程序需要的临时工作空间

每个服务器都需要这么多空间，下面又是一个乘法效应：1GB 的磁盘可能很便宜，但是你得买若干个，每个服务器都要。20 个服务器需要 20GB，而不是 1GB。

另外，不要忘记考虑 RAID 的影响。几乎所有的数据中心类服务器都从 RAID 1 启动。RAID 1 的镜像驱动器，需要 100%的开销，即磁盘数量要加倍。数据往往存储在 RAID 5 上，需要 20%的开销。这又是一个乘法效应。如果你在使用镜像驱动器，就需要双倍的磁盘空间。那么 1GB 现在就变成了 40GB，因为 20 个服务器各有 1GB，而且每个都有镜像。

当你考虑备份时情况就更糟了。20 台服务器上的备份可能带来大量的数据。这将迫使你并行地使用磁带驱动器以按时完成备份，你还可能需要更多的磁带。

1GB 的本地存储成本不到 1 美元。然而，在企业里，管理存储可能会高达 1GB 7 美元。因此，与公司的 IT 团队在设计和生产时讨论存储管理是非常重要的。

管理存储：磁盘驱动器上的空间加上如 RAID 0 或镜像和备份的高可用特性。

小乔爱问……

SAN和NAS是什么，它们有什么区别？

厂商们殚精竭虑地打造各自的品牌，就是为了显得与众不同，结果却听起来差不多，很有意思是吧？SAN和NAS实际上完全不同，但是被这两个缩写模糊了，它们颠来倒去像个回文似的。

NAS（网络附加存储）是指一个存储设备加入到你的IP网络，它通过NFS（网络文件系统）或Windows下的类似协议CIFS来提供空间。NAS设备通常支持多个辅助协议，比如HTTP和FTP，它们都基于一套磁盘和文件系统。NAS设备通常构建在Linux或嵌入式Windows之上，有些也使用其他的类UNIX系统。大多数NAS设备都有多个驱动，它们可配置为RAID 0、RAID 1或RAID 5。

SAN（存储区域网络）是完全不同的东西。一个SAN实际上是区别于你现有IP网络的完全不同的网络。SAN使用Fibre Channel（FC，网状通道）网络，速度可达2GB/s。如果使用SAN，你会看到有个大框架，很多磁盘驱动接入其中。SAN管理软件允许管理员在物理

驱动之上用任何RAID级别来创建逻辑卷。对于客户端操作系统，SAN主机总线适配器（HBA）让远程驱动表现得像本地物理驱动一样。

因为SAN确实是全新且不同的网络，所以它需要很多耗费不菲的基础设施。SAN会立即成为对业务至关重要的东西，因而这个基础设施必须快速、可靠和冗余。SAN的可靠性来自多路径，每个服务器会有多条路径到达每个卷。这意味着需要有双通道卡的服务器、双交换机和存储框架上需要双控制器。

NAS起步代价低，500GB的NAS单元不到700美元，1T的NAS单元不到1000美元。相反，如果是企业级的SAN，至少需要耗费上百万美元。SAN需要随时的系统管理，然而NAS通常成为不需要什么结构和管理的储物场所。NAS是一个部门甚至项目团队就能做出的选择，但是SAN是由CIO级别做出的决策。

带宽很便宜

这样的话虽不常明说，但是大家都是这样认为的。今天的市场上，OC3 连接成本大概在每月 7500 到 12 000 美元。一个像样的企业至少会有一对这样的连接。负载均衡的两个连接，理论上，如果按 310Mb 每秒的带宽计算，每月需要 15 000 至 24 000 美元。

根据流量的曲线，你可能发现用一个突发而非专线的连接更有利。专线连接的价格最好预估，每天每秒你都有同样的带宽，费率也不变。而突发连接，通常花费比专门连接少，只要你流量保持在“承诺”的水平之下，你就只用付固定的费用。

如果你的传输率超过了承诺带宽，你得为此付费[①]。虽说突发连接比专线连接便宜，然而如果你每天的流量峰值超过了承诺带宽，成本可能会迅速上升。如果你的网站被 Slashdot 引用，那就当心你的钱包吧！

那么，一对 OC3 连接有多少并发请求？让我们看些变量。讽刺的是，你的宽带用户越多，带宽看起来就越糟糕。TCP/IP 握手协议保证他们会像接收数据一样快地请求数据。因为他们比拨号用户接收数据快，因此宽带用户会占用更多的带宽。典型的拨号用户依然是 44Kbps。一旦排除了 PPP 的开销，他们可能仅以 38Kbps 或 39Kbps 从网站获取数据。一些电缆用户能够以 6Mbps 下载。稍加计算就能知道，服务一个电缆用户等于服务 13 个拨号用户。

如果大多数用户从其他服务连接，那么景象就如同家庭宽带用户一样。企业网络限制雇员的计算机在 10Mbps，但未来几年你会看到桌面上显示的是 100Mbps。即便很多企业网络现在有 G 级别以太网，他们依然阻止独占网络。而且，企业防火墙/代理服务器会在高峰时间饱和。因此，用户最可能登录网站的时间（主要是午后查看邮件和上网时间），也就是他们最可能因为代

① 就像电力的千瓦时一样：超出的传输率乘以这一级的分钟数。

理服务器负担过重而网速变慢的时间。所以说，企业用户最终和DSL或拨号用户一样使用相同的带宽。

考虑带宽的消耗时，就像考虑CPU和存储一样。你必须看到乘法效应。动态产生的页面会有很多垃圾字符。假设每个页面有1KB的垃圾字符，每天100万个页面，就要发送超过1 024 000 000多余的字节，也就是1G不必要的流量。大多数页面都远超了这个数字。我会在下一章中讨论页面构建技术时讲解如何去掉这些浪费。

8.6 总结

容量管理是一个监视和优化的动态过程，它涉及很多方面，会遇到很多问题。软件变化、流量变化，甚至营销活动都可能大大占用你的容量。考察容量需要从系统整体把握。容量的过度简化或线性模型会误导你，让你的公司消耗大量的钱财或者损失许多收入。

容量是一个系统在一段时间可能产生多少回报的基本指标。因此，设计选择不当会降低容量，也就直接减少了公司的收入。亡羊补牢则需要更多的资金和运营费用。为了让投资发挥最大效益，你需要做下面这些事。

- 总是留意乘法效应。这些会主导你的成本。
- 理解一层对另一层的影响。
- 改进非约束度量不会提高容量。
- 试着在无人等待期间完成尽可能多的工作。
- 设置各种安全阈值：超时、最大内存消耗、最大连接数等。
- 保护那些请求处理线程。
- 持续监视容量。每次应用程序发布都可能影响可扩展性和性能。用户需求或流量模式的变化都可能改变系统负载。

第 9 章
容量反模式

正如你在前一章所看到的，如果 CPU、磁盘、带宽很昂贵，最好你能充分利用好它们的价值。这一章，你会查看一些威胁你系统容量的反模式。

这些反模式使你的应用程序做一些无用功，将电转化为了热而不是收入。硬件很昂贵，因此我们需要物尽其用。

9.1 资源池竞争



我对数据库连接池是既爱又恨。我视它们为不可避免的罪恶。由于要花 250 毫秒来建立一个新的数据库连接，因此确实值得重用这些连接。用得好，它就如同资源池一样能改善吞吐量，从而提高容量。然而，用得不好，资源池会很快成为应用程序最大的瓶颈。

当存在资源竞争时，瓶颈就产生了。多个线程竞争同一个资源，大多数连接池会简单地阻塞请求线程直到有空闲资源。这对吞吐量无益。图 9-1 说明了不同大小的池下，资源池竞争所花掉的时间百分比。线程数小于可用资源数时竞争为零。曲线最高点表现了当线程竞争 4 个数据库连接时的情况。当有 30 个活跃请求时，超过 80%的 CPU 时间都花在了无意义的等待空闲连接。

如果请求处理线程大多数时间被阻塞，处于等待资源的状态，那么很明显它们就没有在处理请求了。的确，图 9-2 表现了针对不同的资源池大小，各个请求处理线程的吞吐量（单位任务/分钟）。这些线代表了不同大小的资源池。在图的左边，每条曲线都差不多是线性的，而且都有相似的坡度。每条曲线在线程数超过资源数时开始走向平坦。这就是阿喀琉斯之踵，做过负载测试的人大概都很熟悉了。很大程度上，一旦资源竞争开始，新的请求处理线程对吞吐量就没意义了。

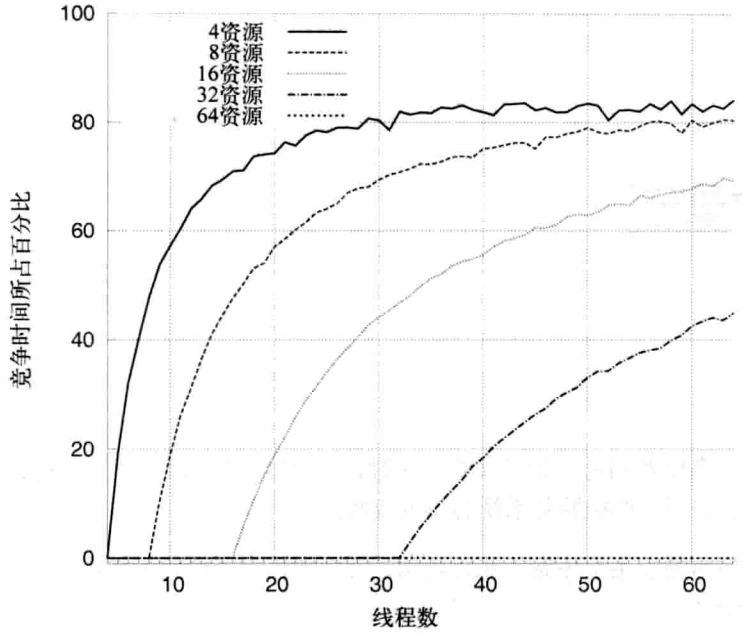


图 9-1 线程间的竞争

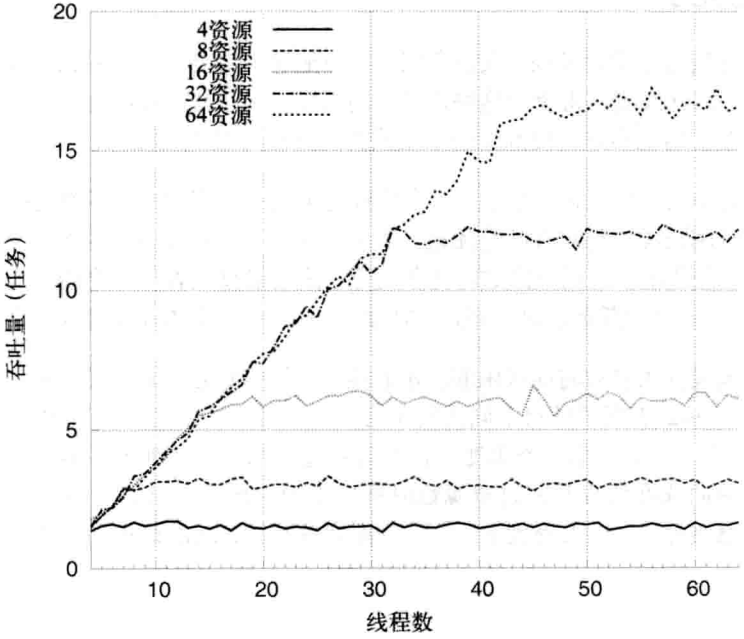


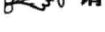
图 9-2 竞争对吞吐量的影响

理想情况，每个线程只要需要就能立即获得资源。为了保证这一点，可以使资源池数量等同于线程数量。虽然这减少了应用服务器的竞争，但是会将问题转移到数据库服务器。比如对于Oracle，每个连接都会在数据库服务器端产生一个进程。根据配置，这些进程会使用几兆的内存。如果有很多应用服务器，则数据库服务器会走向又一个乘法效应。

例如，一个有 20 台服务器的服务器农场，每台机器运行 5 个应用服务器实例，每个实例又使用一个有 50 个数据库连接的连接池，那么数据库服务器必须有 5000 个连接。如果每个连接只使用 1MB 内存，那么数据库服务器就要使用 5GB 内存。更糟糕的是，很多连接在大部分时间都处于空闲。如果它们一直繁忙，那么数据库会吃紧。其他的数据库不会为每个连接产生进程[①]，但是它们一定会为每个连接分配一些资源。

当资源耗尽时，无限期阻塞会带来稳定性问题。（参见 4.5 节“阻塞的线程”。）如果资源池配置为阻塞一个有限的时间，系统能运行更好。每个应用服务器都配置不同，因此对自己系统特别的资源池行为有个专业水平的了解，一定能受益匪浅[②]。当等待资源超时，池要么返回空要么抛出异常。应用程序代码必须处理这种逻辑。

通常的资源池类都在一定程度上不够透明。运行时，你想知道调用者阻塞的频率是多少，启动后最多有多少资源被签出（高水位线），有多少资源被创建和销毁。所有这些指标都凸现了容量问题。JBoss 和 WebLogic 允许通过 JMX 访问一些信息，但是需要你定时地获得这些数据做离线分析。



请记住

- 在正常负载下消除竞争

对于常规的高峰时候，不应该有资源竞争。常规高峰出现在非旺季的时候。

- 如果可能，根据请求线程池来调整资源池大小

如果只要请求处理线程需要，总会有资源就绪，那么就没有效率的损失。对于数据库连接，新加的连接会消耗数据库服务器内存，这虽然昂贵，不过比损失的收入还是要少。但是，请当心单个数据库所能处理的最大连接数。如果存在失效备援，数据库集群的一个节点必须服务于所有的请求和所有的连接。

- 避免恶性循环

资源竞争让事务执行时间变长。较慢的事务产生较多的资源竞争。如果你有多个资源池，那

① Oracle 也支持更加复杂的多线程服务器（MTS）配置，它复用了对多个进程的连接。能适当配置 MTS 的 DBA 们代表了 Oracle 使用的最高水平。

② Jakarta Commons 的 `BasicDataSource` 通过 `maxWait` 属性支持这种行为。JBoss 使用的是数据源配置文件中的。请查看你的文档。

么可能恶性循环，使吞吐量随响应时间增加而呈指数级减少。

- 关注阻塞线程模式

当资源不可用而线程永久阻塞时，资源池竞争的容量问题能很快转变为稳定性问题。

9.2 泛滥的 JSP 碎片



在 Java 领域，JSP 是标准的页面模板语言。JSP 文件采用两遍编译。首先，应用服务器根据 servlet 代码产生一个.java 文件。然后，应用服务器将这个源文件编译成字节码。在第二步编译以后，新的类文件加载进 JVM。像所有的 Java 类一样，被编译的 JSP 被加载进 JVM 持久代。持久代就和它名字一样，其中的对象是真实的类和方法定义。

J2EE 应用服务器几乎总是使用 JVM 参数`-noclassgc` 来启动脚本。这个选项告诉 JVM 不要从持久代释放类。这通常有利于性能提升。当你有大量的 JSP 类被装载时，问题就产生了。这时，默认的持久代大小可能不够大。如果对 JVM 可能加载的 JSP 数量没有限制，则你需要的持久代也就没有上限。由于每个 JSP 被编译并加载进持久代，并且没有垃圾收集，持久代最终会被装满。这时，去掉`-noclassgc` 选项会减轻问题。本质而言，你能够避免逐渐恶化的系统性能，这种性能恶化可能导致服务器崩溃，不过加上这个选项可能使系统始终会略慢些，但是不会产生性能恶化。

请记住

- 不要使用竞争代码

将 JSP 类加载进内存是种缓存机制。如果有足够多的 JSP 文件，你会填满持久代。即便没有，这也是一种对有用内存的浪费，因为在重启应用服务器之前这些内存中的对象可能不会再被访问了。

作为内容的 JSP

我曾遇到过将 JSP 被当作内容而不是代码。我所工作的一个网站有超过 25 000 个 JSP 碎片。它们主要是产品促销的一段内容或者目录级别的页面。问题是它们都没有被释放。服务器一天跑下来，越来越多的内容 JSP 被作为类加载进持久代。这可能使垃圾收集出现严重问题，因为 JVM 会越来越艰难地将所有的类装进持久代（特别是如果你使用`-XX:MaxPermSize` 选项来限制堆的份额）。这时，这些 JSP 甚至不需要可执行代码。它们只是静态内容，因此最好使用 HTML 片段来缓存内容以管理这 25 000 个页面片段。（有关如何合理使用缓存的其他建议，见 10.2 节“谨慎使用缓存”。）

9.3 AJAX 过度之伤



AJAX（Asynchronous JavaScript and XML）几乎已经等同于类似 Google Maps 和 del.icio.us 这样的“Web 2.0”网站了。网页中嵌入的 AJAX 程序会在背地里将请求发到网站，这时用户也能做其他事。服务器的响应被用来更新一部分网页，而不是像“Web 1.0”技术那样每次内容更新都加载整个页面。

虽然 AJAX 技术已经存在好些年，但是 Google 用 Gmail 和 Google Maps 才让它们备受瞩目。这些程序使每个人的注意力放在要做的事情上，而不是已经做的事情。Gmail 能自动补全地址，或在不重新加载页面的情况下扩展和崩塌部分会话。Google Maps 更进一步。你能移动和缩放地图而不重新加载页面。新的地图块异步地加载并显示出来。

不幸的是，并非每个网站都有 Google 所拥有的那么大的服务器农场。对于很多网站而言，AJAX 应用意味着浏览器会向服务器发送更多的请求，而且它们来得更快。平均一个用户在点击页面按钮间有 5 到 10 秒的“考虑时间”。（有可能“考虑时间”的一部分是“等待时间”，让用户等待页面完成下载和渲染。）而对于一个基于 AJAX 的网站，HTTP 请求的时间间隔大约是 1 至 3 秒。个别请求可能更少，因为响应只包括部分页面而非全部页面，响应时间也更少。总体效应要取决于你使用 AJAX 技术的具体方式。用得好，它能减少带宽消耗；用得不好，AJAX 技术会给网络服务器和应用服务器层带来更多负担。

对 AJAX 全面的叙述已超过本书范围（参看 *Pragmatic Ajax*[JG06]中的更多细节），但是由于它是双刃剑，所以有很多使用它的方法。

交互设计

AJAX 是一种技术，而不是目标。用户体验是主要的，因此只在能提高用户交互流畅性的地方使用 AJAX。例如，我曾见到一个网站构建了一个“AJAX 首页”，一个简单只包含 HTML 的主页。网站所有的交互都通过 AJAX 请求，并都显示在同一个页面。这很糟糕。网站似乎要毁掉浏览器！后退按钮会让浏览器离开网站，而不是回退一个页面。更糟糕的是，用户动作后的一小会儿就会发生更新，而这个时间会根据请求复杂性而变，结果页面像是在随机反应，而不是浏览器的通常反应方式了。

使用 AJAX 最好的地方是那些代表了用户脑海中单个任务的地方。以 Gmail 为例，这个任务就是“发送邮件”。通常可能会需要几个页面来完成这个多步交互。如果你的网站有“线框”，你可以查找出一条最终会返回主页或其他连结的线性页面链路。

请求定时

某些早期的 AJAX 库的教程展示了每 1/4 秒自动补全请求，这明显会增加网站服务器的负担。

较新的库可以配置延时，所以它们能在用户完成输入以后的一段时间（通常 500 毫秒）才发送自动补全请求。当然，这是请求发出前的延时。用户感受到的延迟则包括这段延迟，以及发送请求的延时和服务器处理的时间，然后才能获得结果。

会话抖动

请确保配置会话亲缘性，以便 AJAX 请求通过用户会话所在的相同的应用服务器。避免没必要的会话失效备援。

响应格式化

不要返回 HTML 页面或片段。HTML 是没必要的冗余，它会浪费带宽。相反，只返回数据，不要格式，让用户能自动更新页面元素。

对数据使用 JavaScript 对象符号（JSON）[①]，而不是 XML。我知道 AJAX 这个名字里有 XML，但是那只是因为比起“AJAS”来说它更朗朗上口。JSON 非常容易在浏览器中解析，并且较少冗余（可减少带宽消耗）。请参见http://www.json.org/example.html中一些对比 JSON 和 XML 简洁性的例子。

JSON 的黑暗面

关于 JSON 有一点需要当心。有些例子使用 eval()函数来执行 JSON 字符串，因此将对象直接放进解释器域。这是个恐怖的想法！即便你使用了 SSL 和强证书来为浏览器验证这个网站，你该如何对网站验证应用程序代码呢？恶意程序员可以构造一段 JSON 执行在用户机器上。有害的 JSON 脚本可能干很多坏事，比如发送私有数据给第三方，或者提交假命令。当然，“黑帽”程序员干起坏事来似乎无所不能，但是为什么要让他们轻易地攻击你的用户呢？

请记住

- 避免没必要的请求

不要以轮询方式发送请求来实现一些无关紧要的功能，比如自动补全。如果你必须要实现自动补全，比如地址簿、部分数字、部门名称等，你可以只在输入框内容改变时发送请求。（有些在线的教程每 1/4 秒就发送一次请求，太过分了！）

① 参见 http://www.json.org/。

- 尊重你的会话架构

确保你的 AJAX 请求包括了会话 ID cookie 或查询参数。（相对查询参数来说，会话 cookie 更容易！）否则，你的应用服务器会为每个 AJAX 请求创建新的浪费的会话。

- 尽量减小响应的大小

返回必需的数据时数据量要尽量少。返回通过 XML 或者 JSON，而不是 HTML。

- 增加网络层的大小

网络服务器需要应付大量请求。一定要尽可能增加网络层能够处理的连接数，要么添加服务器，要么增加每个服务器能够处理的客户端数。[①]

9.4 驻留过久的会话



Java Servlet 规范最初创建时存在一些晦涩的地方，规定了默认的会话超时是 30 分钟。这是在用户最后一次请求后，会话所能在内存中驻留的时长。第 7 章讨论过，用户在会话超时之初就已经离开了，结果你还在那里痴痴地等待用户最后的请求。（很傻很天真，是吧？）你能确信的是会话会在用户离开很久后依然存在内存中。

只要会话是活跃的，它就会消耗资源，主要是内存。Java 应用服务器总是内存吃紧——很多人称它们是内存猪（一天到晚就知道吃内存）。大量空闲的内存对于 Java 应用程序的稳定性和性能很重要。因此，内存中的会话直接威胁了系统的健康状况。它们对系统的威胁大小正比于它们占用内存的数量。

你的目标是尽早地清除这些会话。默认的 30 分钟超时太长了。对于一个活跃的网站，要监视其流量模式，找出页面请求间的平均时间间隔。换句话说，同一天不同时间的两次访问并不是相同的工作流。同样，相同用户的两次访问，相隔一个小时但都来自首页，也很可能不在同一个会话。反过来说，来自两个页面相隔 29 分钟的请求，却有可能来自用户的同一次活动。一个有效的尝试是将会话超时设置为过去延迟的平均值。实际上，对于零售网站大概是 10 分钟，对于多媒体门户大约 5 分钟，而对于旅游行业网站来说可以达到 20 分钟。

更好的办法是让会话本身可有可无。如果会话包含一些临时状态，那么就让会话保留久一点。另一方面，如果会话中的内容只是持久状态的内存拷贝，那么会话就能被去掉，以后再重新构造。使用这种方法，会话就只是一个内存缓存。我强烈推荐这个模型，因为它既对系统容量和稳定性有利，也对用户有利。只有开发人员了解会话背后的原理，用户并不喜欢受制于时钟。只是因为时钟到期而被踢回起点，是让人极其沮丧的。用户想要的是来去自由，随时继续交互过程。

① 对于 Apache 来说，就是 MaxClients 配置参数。

把会话超时设置为平均的思考时间加上一个标准的时间差。

这意味着不该有只是临时保留的购物车，用户的搜索结果应该予以保留，不管这是使用了搜索引擎还是数据库查询。用户标识和喜好仍然应该活跃。（然而，用户应该能将他使用的设备和他的身份分离，比如在图书馆或飞机场使用的终端。）

唯一的例外是财务方面比较敏感的数据，比如信用卡或社会保险号。[①]

请记住

- 减少内存驻留

合理地让会话在内存中保留较短的时间。这个时间在不同领域是不同的。

- 记住用户不理解会话

用户能理解由于安全理由自动的注销，但是他们不理解为什么购物车消失了，只是因为他们花了 20 分钟在另一个网站上搜索产品。东西不该因为用户只是离开了一小会儿就无影无踪。

- 保存 key，而不是整个对象

如果你保存整个对象在会话中，就使用软引用（soft reference）。用 key 来保存永久对象。这样会有更大的容量，你的用户也会有更好的体验：他们能在一小时后回来，或在一星期以后，然后提取他们留下的东西。

9.5　HTML 中浪费的空间



我曾听过很多谬论说“带宽很便宜”和“CPU 很便宜”。8.5 节我们分析过，这些观点都不属实。你不该为低效找借口。

网络应用程序包含很多无效的东西，它们每天都折磨着用户。假设一个页面能够从 200KB 减量到 150KB（这并非罕见的减量），用户就能少下载 25%的数据。对于拨号连接来说，那 50KB 会花 10 秒钟。而对于宽带用户这节省的字节也很重要。让我给你分析一下。

应用服务器使用 CPU 周期来产生这 50KB 浪费的空间。它们流过应用服务器的网卡时，会消耗带宽。它们经过网络交换机，取校验和，被缓冲、路由，重新取校验和，然后序列化到交换机的其他端口。那个端口可能连接到防火墙（一个专用的路由器），和交换机做着相同的操作，外加一些安全检查。这 50KB 再次回到交换机，可能回到原来的那个，也可能是另一个，然后通

① 社会保险号是美国全国统一的身份识别号。

过同样的低级网络，然后达到网络服务器。网络服务器在内存中缓冲 50KB，和页面的有意义部分。如果每个请求使用 200KB 而不是网络服务器的 150KB 内存，那么网路服务器需要多 33%内存来处理用户请求。一旦整个页面被产生，网络服务器将它发送到浏览器，通过另一个防火墙，交换机，和至少一个路由器。

这 50KB 经过了如此多的地方！每一步，它们要么使用更多内存，要么消耗更多带宽。

还有另一层效应。页面越大，浏览器和网络服务器保存连接开放的时间越长。当连接打开，就没有其他请求可以使用它了。但是，网络服务器只能处理有限的连接！同样线程可能竞争数据库连接[①]，最终用户要竞争网络服务器连接。当用户不能得到网络服务器连接，你的网站也可能会停机。

所以你一定要确保这些连接得到服务，并尽快地释放。

因此，如果明知有害，那么谁还会发送虚胖的 HTML 页面？当然没有人会这么干。虚胖从来不受欢迎，但它躲在没有人注意到的地方。设计人员在 Dreamweaver 中创建了时髦的页面布局。应用程序员将其转变成 ASP、JSP，或者.rhtml 模板。应用服务器根据模板产生所有页面。这个过程中的每个步骤都可能带来无效的浪费。我们来看看最常见的几种类型。

空白

网络应用程序都是由页面片段构成，比如 JSP、JHTML、tiles 或者.rhtml。这些片段让程序员工作更容易，因为它使 HTML 代码能够重用，并且使其从页面上层结构中分离。不利的一方面是将这些页面片段组合到一起可能形成非常大的页面。最容易带来浪费的就是空格。所有这些由不同模版语言定制的标记被产生的内容替换，但是以可读方式格式化文件的空格也被保留了下来。这包括模板标记后的换行，甚至当模板标记什么都不扩展时也如此，比如带错误条件的条件标记。

有个让人大吃一惊的网站，一个标题页面竟然就包括了 100 多个 JSP 片段。产生的 HTML 内容多达 600KB，而 1/3 都是换行符。

听起来似乎没什么，但是想想一个拨号用户需要等 5 秒钟来下载这些空格啊！对系统的影响也是深远的。这些空格在每次页面请求时在网络服务器端都需要占用 200KB，因为它们需要被缓存然后发送给浏览器。由于网络服务器是受限于内存而不是 CPU 的，这会减少网站的整体容量。这些多余的空格会让公司每年带宽成本增加 15 000 美元以上，还不谈网络服务器的其他成本。

空格每年会耗费 15 000 美元。

① 参见 9.1 节“资源池竞争”。

你可以在应用服务器上添加一个截断器，来过滤这些多余的空格。以我所见，过滤器带来的CPU 消耗远少于没有过滤器而产生的内存和带宽消耗。

昂贵的空白图片

几乎每个动态产生的 HTML 页面都有几个空白图片。这些颜色各异的小透明 GIF 或者 1 像素图片无处不在（有时也称为垫片）。它们耗费带宽，降低了用户体验。

假设你有一个表元需要一定的大小来布局，你可能已经设置了宽和高，但是很常见地会看到用空白占位符来做为表元的内容。例如：

```
<img src="/images/spacer.gif" width="17" height="23">
```

这是 53 个字节的图片引用。它能够被下面这几个字节代替：

```
 
```

这 5 个字节替换了 53 个字节，节省了 48 个字节。48 个字节听起来似乎不多，但你要知道，通常一个页面各个角落会有 12~40 个占位符。

别忘了乘法效应。一个页面有 12 处，每处节省 48 个字节，假设每天有 1 百万个页面请求，那就等于 576 000 000 个字节。你的交换机和路由器会每天注意到多发了 576 000 000 个字节吗？

对用户而言，每个不同的空白占位符（并非页面的引用，而是 URL）都意味着给了服务器更多的请求。即便用户浏览器已经有图片缓存，浏览器也会问“这个图片我已经缓存了吗？”。对于动态产生的页面，通常确实会告诉浏览器不要缓存页面内容。

对于空白占位符的每个请求，或检查它的缓存状态，都要占用网络服务器链接一小段时间。网络服务器会无数次发送空白占位符，但其实完全可以把这些功夫用来为用户处理能产生效益的事务。

多余的HTML表格

让我们来讨论 HTML 页面，看看格式化用的表格。

当前所有的浏览器版本都支持层叠样式表（CSS）。明智地使用 CSS 和样式能准确地控制版式，做出表格来。二者的容量差别很大。表格结构必须为每个页面发送，每次页面请求时都要提供。而层叠样式表只用下载一次。

即便样式表比典型的 HTML 页面大，节省也很大。我曾见到过对比 CSS 格式化的页面和 HTML 格式化的页面，CSS 的版本比 HTML 版本的一半还少。对于这样用纯 CSS 和纯 HTML 设计的例子，可以花时间查看一下网站http://www.csszengarden.com/。每次我看到这个网站时，我

都会不止一次地惊叹："我居然不知道还能这样做!"

请记住

- 去掉无意义的字符

在 HTML 中去掉浪费的字符。产生这些字符需要在应用服务器花费 CPU，在应用服务器的网卡上、交换机上和网络服务器上消耗带宽，也会消耗用户的带宽。下载很耗时间，特别是对那些慢连接的用户而言。去掉空格能节省用户时间和公司的成本。

- 去掉空格

空格潜伏在循环、条件、`include` 附近产生的页面中。它们能够被 0 长度的字符串代替，但仍能保留良好的格式。我曾见过有 400KB 的制表符、空格和换行的页面。

- 用不间断空格或 CSS 替代空白图片占位符

网页设计程序会留下空白图片占位符，比如在页面中鼠标下落。很小但是很恼人。将设计的网页转变成代码时，要尽可能地去掉每个空白图片占位符。

- 用 CSS 样式替换 HTML 表格

除了让页面更加"Web 2.0"，CSS 样式表不会像基于表格的方式那样，每个页面都下载。带有样式和类型的 HTML 文件比都是表格的文件要更小。

9.6 刷新按钮



我敢说，用户是系统遭遇的最糟糕的东西。当系统面临压力时更是如此。假设你的网站在变慢。如果这是一个 J2EE 网站，那么一些用户有时会让垃圾收集器装满，使它需要 15 秒才能响应。（我发现很滑稽的是，由于大多数垃圾收集发生在内存分配失败时，因此在处理某些用户请求时它们保证会发生。）坐在浏览器前的用户不知道在网络的另一端到底在做什么或者不在做什么。如果用户在 10 秒内没有看到页面，他很可能就要点击刷新按钮。这时，浏览器放弃了前一个请求，打开新的套接字，发起新一轮的 HTTP 请求。

没有人告诉应用服务器要停止处理前一个请求。实际上，如果网络服务器和应用服务器缓存间的连接器响应了，甚至不可能发现前一个连接已经被放弃了，除非整个页面已经构建并缓存到网络服务器内存中。最好的情况下，当尝试发送响应时，应用服务器获得了 `IOException` 异常，但这仍然是一个少见的理想情况。

如果页面请求导致事务发生时，比如更新用户的配置或者对跟踪数据做日志，那么第二个请求可能会阻塞等待第一个的完成。糟糕的是，它甚至可能因第一个请求而死锁。

对于刷新按钮没有什么好的解决办法。只是请确保你的网站足够快，用户也就不会总去点击它了。

请记住

- 让刷新按钮成为一个摆设

快的网站不会迫使用户点击刷新按钮。你需要让网站对页面处理得足够快，使刷新按钮没有用武之地。刷新请求会让网站雪上加霜。

序列化源地址

我见过一些应用服务器每次从一个源 IP 地址只处理一个请求。这可能在两个方面无效。首先，如果你使用诸如 Akamai 这样的缓存网络，所有的请求似乎都来自同一个 IP 地址。如果用户来自的企业网络只有一个网关代理服务器，也是如此。其次，一旦用户点击刷新按钮，用户的浏览器就不再等待第一个请求的响应。序列化请求会恶化这个问题，因为用户的第二个请求不会在第一个完成前开始处理。这意味着这个沮丧的用户会一直看到这个转动的图标而不是页面，他会不停地点击刷新按钮，然后愤然离去，投奔到竞争对手的网站。

根据你的负载平衡和会话亲缘性配置，第二个请求甚至不会去找第一次请求同样的应用服务器。因此，你的应用程序代码需要为单个用户做好准备，在不产生死锁的情况下数次处理同样的事务。

9.7 手工的 SQL 语句



现在每个应用程序都要用到关系数据库。很多应用程序要依赖对象/关系映射来搭桥。不管你使用 Ruby on Rails 的 `ActiveRecord`，还是 Hibernate，或者三种 EJB 持久模型中的任何一个，都要使用对象/关系映射。ORM 包产生可预测的重复的 SQL 语句。SQL 可预测的访问模式对容量有利，因为能干的 DBA 能调优数据库使这些查询执行良好。[①]

这些 ORM 层让开发人员以各种方式着手于直接的 SQL 访问。例如，Hibernate 能让你直接嵌入命名的 SQL 查询到映射文件中，你能在应用程序中通过名称来引用它们。即便它只是访问数据库连接并直接提交查询，也总可能对数据库发送手工的 SQL 语句。这种能力通常被说成是一种提高性能的方式，那么我为什么称它为容量杀手呢？这主要是因为面向对象的开发者对完全无辜的数据库做着各种奇怪、奇异和笨拙的事情。

① 但是请当心 ORM 工具的“N+1”问题。这个问题出现在访问集合成员时。一些 ORM 框架发送查询来查找集合的成员，然后对每个成员分别发送查询来填充单个对象。

问题是开发者手工编制的 SQL 通常是惯用的和不可预测的。对于应用程序其他部分的所有数据库调优都对这些无效，并且还可能损害它们。同样，这些怪异的调优对于应用程序其余部分也没好处，反而可能有害。

是什么让手工 SQL 查询变得很糟糕？它们通常受累于一些常见的错误。第一，它们通常会连接（join）非索引列。第二，它们通常会连接太多的表。如果它只是一个简单的关系，开发人员会只使用 ORM 包来遵循关系。由于他们使用了手工 SQL，那就会有不同寻常的访问模式了。第三，开发者尝试将 SQL 看作要么是过程语言，要么是面向对象的语言，而不是它实际上的基于集合的关系语言。他们通常提交连接了 5 个表的查询来选择一行数据，然后他们会提交这个查询 100 次来找到其他行。第四，开发者喜欢使用 SQL 不该有的功能。我曾经看到一个 8 路联合（union），其中的每个子查询都连接了 5 个非索引列的表。它花了一整页打印所有这个东西。这个查询计划扫描了 40 个表。

表扫描：是在大表中查找的最慢的一种方式。数据库服务器会对表中的每一行进行迭代查找。

正确调优的数据库对于产品使用很重要，有时甚至达到难以置信的效果。每个 DBA 都有只是添加了一个索引或分析表统计，就将 18 个小时减少到 3 分钟的故事。提高一个数量级是很常见的。开发者看不到这些效果，因为他们在开发和测试中，使用的是不现实的小数据集。性能需要真实的数据规模。有时候，已存在的产品系统能提供可用的数据。在它们能用在非产品环境前可能需要擦洗一些。擦洗通常需要用随机数据替换私人数据来保护保密的用户数据，特别当测试数据要发送到公司以外的地方。对于新的开发，不可能有现成的数据集可用。这时，花一到两个小时产生数据是值得的。

手工的 SQL 和动态产生的 SQL 紧密相关。用 ORM 工具动态产生的 SQL 就够了，因为它们可预测。这不是那种我警告你要留意的动态产生的 SQL。那种 SQL 让我在半夜惊醒，比如一些代码通过遍历各个属性并且添加些 WHERE 子句来按实例查询，或者数据表和连接需要动态组装的地方。保留这样的 SQL 给报表系统，而不是事务系统。

每次这样的查询都有触发慢条件的风险。你能调优数据库来对某些可预测的访问模式响应良好。但你无法让它对每个可能的查询都响应良好。

请记住

- 最小化手工 SQL

你可能倾向于写自己的 SQL 来做性能优化。是否有办法让数据库自己通过 hint、索引或视图来做到这些呢？

- 看看 DBA 是否会嘲笑这些查询

如果它过不了这一关，就不该投入到生产中。

- 根据真实数据验证

要对产品规模的数据尝试手工 SQL。开发数据库上看到的好处，有时候在产品环境的查询计划中并不能体现出来。

9.8 数据库富营养化



生态学中，湖泊若经历超营养化就会“衰老”。所谓超营养化是一个缓慢的淤泥积累过程，淤泥中则是死亡的微生物、腐烂的鱼和海藻。发展到后期，这些淤泥消耗了水中大量的氧气，湖中生物无法存活了，于是湖就死亡了。同样的事情发生在你的数据库中，系统膨胀也会导致数据库死亡。

在开发和测试时，测试的数据集是很小的。加上 QA 的时间，数据库架构（shema）很容易未经过严格的高容量测试就进入产品环境。一旦用户开发倾倒数据的时候，一切就开始快速地变化。

索引

ORM（对象/关系映射）工具很容易在表间创建和遍历关系。ORM 工具的映射文件定义的连接不会由 DBA 来审查。因此，尽管数据架构师定义的引用完整规则有可能对外键关系创建索引，通过对象属性关联的表很可能不会获得同样仔细的关注。通过非索引列选择一行，或者说遍历一个对象关系，导致了可怕的“表扫描”。也就是对表中的所有行进行线性查找，寻找某列符合查询条件的行。

由于开发环境所使用的数据集规模，你甚至不能测量带索引列和非索引列查询的时间差异。（确实，如果数据集很小，表扫描可能是查询计划中最快的！如果整个表都已经缓存到内存中，就尤为如此。）但是假如是生产环境中经历了一两年累计的数据，你的用户就算想完成一些简单的操作可能也得等好几分钟。通常，ORM 映射中的关联目标列都应该被索引。

数据库架构通常设计良好，以便于应用程序代码使用这个架构。结果，设计时创建的索引可能不匹配应用中实现的真实访问模式。数据库架构师应该参与到开发中去，确保架构会持续适应应用程序的功能。在敏捷项目中这点特别重要，实体间关系大起大落的变化会远远超过传统的“前期大设计”项目。

切分

另一个关键的设计考虑是要能在生产操作中保持数据库良好的结构。数据库架构师总是想尽可能多地知道表的增长率和每张表期望的数据量。这个信息能帮助他们布局磁盘上的数据表。将

逻辑表映射到物理存储上对于数据库的长期性能有重要影响。不幸的是，早期对增长率和保持率的猜测通常证明是错的。有时候，从一次发布到下一次发布之间也会差别巨大。例如，我曾见到一个订货管理系统从一次发布时一年 1GB 的审计日志，变化到下一次发布时一天 1GB 的日志。这时，数据库的存储布局随着数据表扩展到多层物理区域而变得“笨重”。由于数据库执行了更多的磁盘 I/O，响应时间也增加了。一种解决办法是程序开发前就充分认识到这一点，并采用完美的规范。

另一个解决办法是分割数据表。（叫作 segment 或者 partition，不同的数据库厂商使用不同的术语。）如果有些列有少量的数值，而每个值都指定了一簇相关的行，那么这个列就是切分的好目标。例如一个订单表，处理订单时会翻来覆去地倒腾这张表。通过添加一列“星期几”，数据表能切分成 7 份。每一份都能被重建或者移到其他物理区域上，即便数据表的其他部分使用很频繁。这还能让表的存储重新构造而不让系统停机。

历史数据

由于数据库中数据的增加而导致速度变慢，这常常是因为存档和删除过程变得艰难了。我们面对的是个交易系统。数据都是需要处理的用户交易。这意味着你没必要让所有的在线历史数据都保存在同一个数据库中。特别是，报表和特设分析（ad hoc analysis）不该放在产品数据库中。（一种做法是确保只显示过去 90 天或 6 个月的数据！）

数据挖掘、报表或其他的分析应该放在真正的数据仓库中。这种 OLTP 架构对数据仓库无益。

OLTP：联机事务处理。是为快速插入事务记录而做的架构优化。对于产生报表或执行特设分析，通常很不好用。

用户可能需要或想要看到历史数据。例如，Amazon 现在有我过去 10 年的订单。（我不知道他们计划保存这些订单多少年。）1998 年春季以来的订单多长时间会需要一回？多级存储可以在低成本系统里在线保存历史数据，而在高性能系统中保存了最新的交易数据。

这一节只是谈及了数据和存储架构的皮毛。这个话题值得用一系列的书来讨论。最重要的是：数据清楚严格的体系对于系统的长期稳定和性能至关重要。

不要把事务处理和生成报表混在一起。

请记住

- 创建索引——这不只是 DBA 的责任

你比 DBA 更了解程序的意图。你应该知道哪些列用来查找，哪些表通常用来读，哪些表通

常用来写。应该由你来提出索引的第一次迭代。

- 清除淤泥

旧的数据会拖慢查询和插入。除非用户关心（比如订货历史表），否则你不该将它放在产品服务器上。

- 让报表远离生产

报表数据应该被存在别处。不要让繁重查询任务的报表操作影响生产性能。此外，报表最好使用星型架构，而不是 OLTP 架构。

9.9　集成点延迟



与其他系统的通信都会带来延迟。远程调用比本地调用要多花 1000 倍的时间。不管是通过 DCOM、CORBA、网络服务，还是二进制套接字协议，远程调用的调用者在等待响应时会什么都不做。调用者必须处理一些事务，否则它不会调用。因此，你知道调用者至少花掉远程系统的响应时间。

集成点延迟可能会是严重的性能问题。当集成点使用了一些远程对象协议时，问题就变得敏感了。远程对象的“本地透明”流派认为，调用者不用关心远程调用和本地调用的差别。这条思路不可取，因为以下两个重要的理由。首先，远程调用展现了与本地调用不同的故障模式。它们容易遭受网络故障、远程进程故障、调用者和服务器间的版本不匹配，等等。其次，本地透明使开发者像本地调用一样设计相同的接口，导致了繁琐的接口。这种设计对单个交互使用多个方法调用。每个方法调用都受到延迟的折磨。累积起来的效果是响应缓慢。

个别用户的性能问题对整个系统带来了容量问题。即便处理事务的线程在等待响应时暂时是空闲的，它依然占用了很多资源。它会消耗内存和 CPU 时间片，它也可能占用数据库连接。进一步说，空闲线程可能在数据库中产生行锁或页锁，从而导致竞争。

抽象点来说，就是让线程阻塞等待来自集成点的响应有一个机会成本。它不能干其他的工作，即便这些工作进入队列在等待线程处理。

请记住

- 尽可能少地遭遇延迟

集成点延迟就像十三点赌局中的庄家优势。你玩得越多，遇到麻烦的机会也越大。请避免繁琐的远程协议。它们执行时间较长，而且束缚了宝贵的请求处理线程。

海外来的 RMI

我曾经参与了一个胖客户端 Java 系统，系统使用了 RMI 来对服务器通信。服务器提供了功能丰富的企业数据仓库的面向对象视图。客户端应用程序允许营销人员让某一层的产品与媒体数据进行连接，这些媒体数据有图片、PDF 文件和文本文件。

在国内，系统工作良好。可是后来有了一些英国的用户，他们嚷嚷着要做本地服务器和数据仓库，这可是数百万的投资啊！我们发现扩展层级结构中的一个节点（一个树型控件），美国用户 1 秒钟都不到就可以搞定，但是英国用户却要花 20 多分钟。

树型控件中扩展一个节点需要客户端查询节点的子节点：它的名字、类型和孩子数量。问题是客户端会对父节点做远程调用，查询孩子集合，然后客户端会枚举这个列表，对每个孩子调用三次。这是一个“1+*N*”问题的例子：对列表的一次调用，要加上对列表中每个元素的一次或多次调用。

我们给父节点添加了一个方法，返回树型控件需要的 3 比特信息的“总结对象”。用了这个更新后，英国用户也能像美国用户一样享受到少于一秒钟的响应时间，而不用对整个数据仓库安装一个镜像！

9.10 cookie 怪兽



有些技术产品误导你滥用。拿冲天炮来说，无论贴有多少个警示标签，也都还是带手柄的爆竹。问问交互设计大师 Don Norman——手柄意味着要被拿着。[①]就算是个一辈子也没见过鞭炮的人，几分钟内他也会拿着冲天炮来放。这很不安全，但是血的教训人们总是从头学起。

HTTP cookie 就在那里，跟冲天炮一样，都属于“请你来自杀”式的产品。它们有着合情合理的用途，但是一旦滥用会让你痛苦不堪。表面上，HTTP cookie 似乎是种无害的、只是 Web 交互无状态环境下有点不优雅的不成熟产品。服务器发送一两个附加的首部，带上一点期望客户端在未来会发送回的数据。RFC 2109[②]的原作者清楚地打算将 cookies 用于对话管理。事实上，RFC 2109 被称为“HTTP 状态管理机制”。David Kristol 和 Lou Montulli 不仅仅是为对话标识符添加一个新的 HTTP 首部，他们发明的 HTTP cookie 应用更加普遍。HTTP cookie 能够用于对话标识，也能用于更多事情。（Tim Berners-Lee 做了首批好的选择，即让 HTTP 通过首部进行扩展，要求代理忽略它们不理解的首部，而不是抛出错误。）

一些开发者已经独立地发现了通过 cookie 存储匿名持久数据这个反模式。例如，增加一个拦截器完成请求处理管线，使用 Java 序列化来保持匿名用户的购物车，像 cookie 一样。当然，这

① 参见 Don Norman 著的《设计心理学》。

② 参见 http://www.ietf.org/rfc/rfc2109.txt。

个方法的目的是避免为那些可能永不再回来的匿名访问者产生数据库记录。有了购物车 cookie，匿名用户回来后，还能获得上次访问时的购物车里的物品。

这个方法有很多问题：它很难知道从哪里开始。首先，Java 序列化就会出现一些问题。浏览器会一直保持 cookie 直到失效为止（或者直到用户电脑升级，安装一个不同的浏览器，或者清除浏览器的私人数据，等等）。

结果序列化结构的购物车对象可能有数月或者数年之旧。一处代码的改变可能早在用户返回之前就让其序列化的结构不可用了。反序列化 Java 对象的错误被抛出，如 `IOException`，许多开发者视其为不可能的事情。序列化使 `IOException` 变成日常工作的一个例行程序。

即使源代码相同，用户购物车里的产品可能也很不同——或者会遗失。现在拦截器既要检测版本控制问题，也要处理参照完整性。

那么如果浏览器撒谎了呢？HTTP 与许多协议一样，只是一个关于两个部分应该如何合作的协定而已。合作的任何一方都能被暗中破坏或恶意攻击。如何能保证一个恶意用户无法通过改变 cookie 来设置所有商品的价格为 1 美分？一个足够老练的攻击者，如果熟悉公司的软件平台信息，很容易用个序列化的购物车对象（`Cart`）来破坏价格和促销等许多商业规则。即使这些 cookie 只是数据，也别相信它！很多工具让用户围绕着 HTTP 的请求和响应团团转（Firefox 扩展组件，智能 HTTP 代理，会话重放，等等）。

然后还有容量问题。Cookie 意味着发送大约 100 字节的小块数据。这是你进行对话标识的全部所需。这么小的数据，浏览器为每个新的 HTTP 重新发送 cookie 就无所谓了。直到产生一个 4KB 序列化对象或者 XML 段发生冲突，带宽开始增加。这也是极大的浪费。对于像序列化购物车这些方法来说，你需要反序列化购物车一次。然后，它就会存在用户对话的记忆当中。每次请求都发送和重新发送数据，会消耗你的带宽。

在接受一个请求的时候，网络服务器一定会用它自己的内部结构解析所有的头部。如果有一个应用服务器，则网络服务器向应用服务器重新发送请求参数。要使大家都得到通知，附加的 cookies 总共要两次通过线路（有时是四次），两次被解析。再次说明，单独的一个请求可能不会耗费太多的额外时间，但是我们担心的是乘法效应。

它也伤害了用户。即便是宽带用户，一般使用的上行流量也很少超过下行流量。上传的话，每次请求多 4KB，也会聚少成多。（这里几毫秒，那里几毫秒，然后很快你要考虑能否实时了。）

所有这些附加数据处理在生产环境中形成了一个持续的运行成本——日复一日的耗费。控制 cookie 的附加程序有一个持续的维护开销，这些开销都是为了避免为匿名用户在购物车的数据库里增加数据行。反对将购物车放入数据库，唯一的考虑就是定期清除无用购物车的工作的成本和复杂性。

作为一般的机制，cookies 产生了一些较酷的效果。只要记住客户端会撒谎，可能返回陈旧的或者已破坏的 cookie，也可能根本就不返回 cookie。

请记住

- 提供小 cookie

使用 cookie 来标识，不是存放全部对象。保持对话数据在服务器上，使其不能被恶意客户端改变。

9.11 总结



统计表明大部分程序员从来没有真正做过大规模的、关键任务的软件工作。首先，薪水调查显示大部分程序员的持续工作经验不到十年。一旦工作了十年，很多程序员又转入管理岗位或者干脆不再做编程工作。其次，如果你将项目大小用直方图来表示，那么会发现大多数的项目是小项目。所以，对于这些没有很多机会接触大型项目的年轻人，毫无疑问，高水平的经验是很匮乏的。

没有那些经验，程序员们似乎会因为忽略或者误会，而重新制造出本章提到的很多容量杀手。这些问题肯定不会在大学里讲，那里的最优化一般只涉及调整一些搜索算法。

容量杀手看似明显的错误，也许只有把它们从系统中抽出检查的时候才觉得明显。然而，我看过不同的开发团队在不同的客户端反复生成容量杀手。没有人会故意做出损害系统容量的设计，相反，他们只是在设计选择时没有考虑到对容量的影响。

第 10 章
容量模式

C.A.R. Hoare 有句名言："过早的优化是一切祸害的根源。"这句话经常被误用来当做粗劣设计的借口。Hoare 的原话是："我们应该忘记很小的效率，比如在 97%的时间里不用去想它：过早的优化是一切祸害的根源。"他真实的目的是反对用大量复杂性和开发时间去追求蝇头小利。

问题是把优化工作放在后面考虑经常就意味着"完全没有优化"，时间只会越来越紧张（什么时候不紧张呢？）。进一步说，优化只能把单个函数的性能提高一个百分比，但是不能从根本上带来全新的设计。你绝不会把冒泡排序优化成为快速排序。不想犯过早优化的错误的话，就应该在一开始选择一种更好的设计或者一个为扩展优化了的架构，这样就不用优化了。

同样，设计分布式系统的时候忽视性能和容量会造成极大危险。在 8.5 节"容量的神话"里我们讲过，CPU、内存和磁盘的消耗稍有增加，就会在你的系统投入生产的时候被乘法效应放大到一个非常庞大的数字。我曾经看到过这种低效的代码让一家公司额外增加 1 千万美元的硬件预算，只是为了应付一个假期的旺季。其实，只需要解决几个反模式，应用本章介绍的两个容量模式[①]，就能让他们避免这 1 千万美元的开销。这些花费相当于将近一个星期的销售量。

避免反模式是重要的一步，但是仅仅避免自残是不够的。本章的容量模式将会让你的应用程序从"搏击取胜"升级到"坚如磐石"。

10.1 连接池



就像 9.1 节的反模式"资源池竞争"指出的，资源池可以极大地提高性能。资源池可以消除连接建立时间。建立一个新的数据库连接需要一个 TCP 连接、数据库认证，以及建立数据库会话。这些操作很容易就花掉四五百毫秒的时间。为创建数据库连接所花的时间，几乎都跟启动一个新线程差不多了。

① 即"预计算容量"和"谨慎使用缓存"。

在最初使用 Perl CGI 脚本编程的时期，一段脚本通常打开一个连接，执行一些任务，然后关闭连接。这种策略安全、清晰，并且易于调试。然而不幸的是，当数据库服务器在管理连接上花费的时间和在处理事务上花费的时间一样多时，这种策略就行不通了。为了绕过这一点，单独的页请求必须共享或者复用连接。目前，几乎所有语言和编程模式都支持连接池。实在是没有借口不使用连接池，除非你用不好。

使用连接池确实需要考虑周全。连接可能会出问题，那么，任何想要使用这个连接的请求都将出错。不良连接将被签出，产生错误，然后被扔回到连接池中。状态良好的连接用来执行实际工作，因此相比之下，它们被签出的时间也更长。结果，当请求到来时，不良连接被取出的可能性更大，因此将会导致数量不成比例的错误：十分之一的不良连接导致的请求出错不止 10%。

连接池大小是一个至关重要的问题。空间不足的连接池导致资源池竞争（参见 9.1 节），而过大的连接池将给数据库服务器带来很大的压力。

一个相关的问题是针对将连接签出和签入连接池的设计。对于一个基于网络的系统，有几种可行的策略。最简单的就是“单页”模型：为整个页面签出一个连接，当页面完成时再签入。在涉及多个连接池时，该模型更不容易造成死锁，因为可以为所有的请求施加相同的签出和签入顺序。换句话说，该模型需要更多的连接来应付请求处理线程，因为每个连接都会被签出较长的时间。

“单页”模型通常结合一个事务管理器来实现，后者用于创建一个包含整个页面的事务。

如果单个页面是从不同片段中动态构建的，再使用“单页”模型就不太可能了。“单段”方法允许每个片段签出属于自己的连接，执行一些任务，然后把连接签入连接池。相对于“单页”模型，这种模型更有可能死锁，但是却能达到更高的吞吐量。每个请求处理线程需要的连接更少，因为单个连接被签入得更快。它的主要好处就是，单个片段不需要理会整个事务上下文，每一个片段都可以独立操作。

一种混合方法就是让每个片段管理自己的连接，但是为整个页面创建一个数据库事务。这些连接附属到事务上，直到事务被回滚或者提交。这种模型结合了“单页”模型避免死锁的安全性的优点，以及“单段”模型的简单、隔离性的优点。它像“单页”模型一样需要较大的连接池。这种混合方法很难调试，因为单个片段可能碰到未提交的数据，它们是由前一个片段引入到事务中的。结果是这个片段能够看到的数据你看不到，因此很难去诊断该片段的行为。

不管你的应用实施的是哪种策略，你都必须监控连接池竞争问题，否则，这个容量增强方法很可能快速转变成容量杀手。



请记住

- 连接池

连接池是基本的技术，没有理由不使用。

- 保护请求处理线程

不要让调用者永远阻塞。一定要保证任何调用都有超时时间，当没有归还连接时，调用者要知道该怎么办。

- 根据最大吞吐量设定连接池容量

容量不足的资源池导致竞争，并且增加时延。这首先就违背了使用连接池的初衷。监控对连接池的调用，看看线程要等待多长时间来签出连接。

10.2　谨慎使用缓存



对于性能问题，使用缓存可能特别有效。缓存可以减少数据库服务器的负载，并大大减少响应时间。然而，如果使用不当的话，缓存也会带来新的问题。

应用程序这一级所用缓存的最大内存应该是可配置的。如果不限制占用的最大内存，缓存将最终吃掉系统所有的可用内存。这时，为了处理请求，垃圾回收器将花越来越多的时间来试图恢复足够的内存。缓存消耗了其他任务所需的内存，将最终导致系统运行极其缓慢。

不管为缓存设置的内存大小是多少，你都需要监控缓存单元的命中率，以观察是否大部分的单元都读自缓存。如果命中率很低，那么缓存就没有带来性能上的优势，并且可能比不用缓存时还要差。把物体放到缓存中就像是在押注，你赌的是对象只生成一次的时间加上散列和查找的时间，比每次都生成所用的时间要少。如果某个缓存对象在服务的生存周期中只使用一次，那么缓存它就是多此一举。

避免缓存易于生成的对象也是明智的选择。我曾经见过一次内存缓存包含了几百个只有一个空格的缓存项。那是一段 JSP 中有这样一个条件句，通过在用户资料中查找一个布尔标记判断用户是否是雇员。大部分情况下判定都是假，并且不作渲染。我怀疑一个布尔条件句真有那么耗时吗，以至于需要缓存结果，特别是当缓存对象只和一个用户相关时。

在 Java 中，应该用 `SoftReference` 对象建立缓存来保持缓存单元。如果内存不足，垃圾回收器将保证回收任何只能通过软引用才能访问的对象。结果就是，使用软引用的缓存将帮助而不是阻碍垃圾回收器回收内存。

在极端情况下，可能需要采用多级缓存策略。在这种策略下，你将访问最频繁的数据放到内存中，而使用磁盘来作为二级缓存。当缓存数据非常大（比如图片）或者工作集合比较大而不能放到内存中时，这种方法很有效。如果获取非缓存数据需要访问 WAN 连接，多级缓存也会有帮助。

预计算结果能减少或者消除缓存的需要。

最后，任何缓存都有脏数据的风险。任何缓存都应该有一个验证机制，在对应的源数据改变

时从缓存中删除缓存项。你选择的策略可能对系统容量有重大影响。例如，当有十个或十二个应用服务器时，点对点的通知机制会很有效。如果有几百个应用服务器，点对点的单播就不再有效，你需要寻找消息队列或者其他广播通知的形式。当然，使用广播，还需要确保应用服务器不要同时重载无效的数据而冲击数据库。

请记住

- 限制缓存大小

无限制的缓存将消耗掉那些本来应该用于处理请求的内存。把你曾经载入的每个对象都保存在内存中，对用户并没有好处。

- 建立刷新机制

无论是基于时钟、日历还是网络事件，任何缓存都需要时不时刷新。然而，刷新缓存可能是个耗时的操作，所以要限制缓存刷新的频率，否则你会被自己黑掉。

- 不要缓存无关痛痒的对象

不是每一个域对象和 HTML 片段都是值得缓存的。很少用的、微小的或者不费时的对象都不值得缓存：记账成本和可用内存减少，你付出的代价高于获得的性能改善。

- 考量访问和修改的频率

不要缓存那些可能在使用前就会修改的东西。

10.3 预计算容量



作为应用架构师和设计师，我们喜欢网页上的动态内容。一种原因是，它比静态内容有趣得多。另一个原因是，程序员不需要输出静态内容。我们甚至为它起了一个贬义的术语——公司网页（Brochureware）。当需求提到内容应随时改变时，我们就会想到动态内容，并立刻设计出数据库驱动的、动态生成的网站。所有常见的技术都把大家往那个方向赶：JSP，ASP，Ruby on Rails，等等。[①]

问题再一次来源于乘法效应。考虑一个典型的零售网站。几乎所有的零售网站都允许顾客以某种层次结构浏览产品分类。不管这些分类表现为一个单根树、森林还是一个有向无环图，它们最终都会向用户输出某种形式的产品明细页。至少最上层的分类在每个页面上都会显示。通常第二级的分类也会显示。生成分类菜单的通常做法是在数据库中查询要显示的分类，然后在一个页片段遍历这些分类，对于每一种分类，使用 HTML 或者 JavaScript 渲染。这段代码可能每天运行

① 更不要说 Struts，Tiles，Tapestry，WebWork，WebObjects，SpringMVC 等了。

上百万次。最上层分类多长时间改变一次呢？每三个月？小改变可能每周会有一次。但是那段代码依然执行一次查询，并动态生成 HTML，只是担心万一在上次生成 HTML 后的几十毫秒内分类已经改变了。尽管你缓存了数据库查询结果，因为涉及到为数众多的字符串，渲染 HTML 依然会花费较长的时间。

为什么要花费时间去渲染 HTML 呢？如果你能确定网站中某些章节内容的改变频率远低于生成页面的频率，那就值得预计算渲染的 HTML 片段。当你能够精确确定内容改变的时间时，这尤其有价值。在那时候再重新生成预计算的内容，然后像原来一样使用，而不是上百万次地重新计算同样的 HTML。

像 Slashdot 和 Fark 这样的新闻门户网站会预计算出它们的主要页面。它们的新闻最多按照小时改变，有时更频繁些。评论则每分钟都会完全改变。尽管如此这两种站点的主要页面都是一个模版，装入很多预计算的内容片段。通过每隔几分钟重新计算内容，站点显得很实时的样子，但其实在下一次更新之前，页面的每个版本都被浏览了成百上千次。

脏词屏蔽

第 7 章讨论过的零售网站发布的例子，是我见过的包含过多动态内容的最糟糕的情形之一。垃圾回收的追踪统计显示，每个页面请求创建了近 10MB 的垃圾。现在的 Java 垃圾回收器已经比 Java 2 和 Java 5 虚拟机改善了很多，真的居然有 10MB 吗？那就是代码中出现了严重的问题。等我找到原因，几乎是哭笑不得。我找到了那个开发者，朝他大吼“0x7f”。

每一个产品明细页面，和每一个“product insert”（在主页和分类页的小按钮），都使用了一个叫作“脏词屏蔽”（Profanity Masker）的自定义组件。这个小东西是一个像自定义 JSP 标签的 ATG droplet。一个 droplet 处理自己的内容并生成输出。droplet 的输入在 droplet 被调用前被缓存，输出在它被包含到页面前被缓存。脏词屏蔽被用在产品名字、简短描述、大段描述，和规格说明中。比如音乐，每个标签名上都用。比如电影，每个演员的名字上都用。一个简单的产品明细页面可能有 20 多个这样的小 bugger。

脏词屏蔽使用字符串分析（StringTokenizer）分析内容，然后对照它的有十一个脏词的敏感词列表，逐个比较内容中的词语。如果不在列表中，它就把词语追加到为输出创建的 StringBuffer 中。如果出现匹配，它就只保留首字母，后面的每个字母都用星号替代，这样每个人都能够猜出原始的词语是什么。

花一分钟时间想一下，文字内容的每比特都要被分析，和一个包含坏词语的 `Vector`（是的，不是 `ArrayList`）逐词比较，然后再合在一起。每个页面都要做 20 次，每个页面一天可能被访问 500 多万次。对于每一个页面请求，这将是非常多的字符串比较和 10MB 的垃圾。所有的内容都是晚上发布的。难道脏词语会有机会在白天自然出现吗？为什么要对每个页面浏览都做全部的工作呢？

最有趣的是后话。我们询问是否可以停用脏词屏蔽时（这会污染年轻人但却保护了网站），负责内容的商业发起人知道后暴跳如雷。产品描述、唱片名字、歌词和歌名都是受版权保护的材料，是由数据供应商聚集的，但是最终来源是唱片公司和电影工作室。根据合同，你绝对不能修改这些内容，尽管你有崇高的理由这么做。听到这些内容显示在网站上时被篡改，他十分不安。于是我们删除了脏词屏蔽程序，但是到了后来我们也没有发现为何一开始会使用它。

预计算内容确实要付出些代价。它需要为每个预计算的内容分配存储空间。把标识符映射到文件再读取文件需要运行时开销。对于常用的内容，这种开销可能会促使你把内容缓存到内存中。生成内容的开销主要发生在内容改变的时候。如果在改变前内容已经使用了很多次，那么预计算就是有价值的。

个性化不适合预计算。如果整个页面都是个性化的，预计算内容就是不可能的。在另一方面，如果只有小部分片段或者章节是个性化的，那么页面的主体部分还是可以预计算的，个性化内容部分可以空着。

预计算页面的大部分，个性化内容部分可以先空着。

预计算内容和缓存是不同的。把数据库结果还存在内存中是一种折中，增加一个变量的压力（应用程序服务器的内存）以减小另一个变量的压力（数据库服务器的 CPU 或者数据库 I/O，根据系统的情况选择）。将页面片段还存在内存中也是类似的折中，是在应用服务器响应时间和内存需求之中取得平衡。然而，第一个问题是在内存中缓存页面片段将会最终损害响应时间。如果内存不足或者缓存大小小于片段集合，那么应用服务器将花费时间修改缓存。更糟糕的是，它还得用这么少的内存去服务页面请求。在老一代的堆空间中有数兆的缓存内容，而导致没有足够的内存时，Java 应用服务器会运行得非常、非常慢。第二个问题是，常驻内存的缓存需要时间来“预热”。无论谁从冷缓存的服务器中请求第一个页面时，都得等待好几分钟才能获取一个页面。常驻内存的缓存有它的用武之地，但是存储渲染的内容巨大的页面片段不是使用缓存的正确方式。

预计算内存不是一个“要么全做，要么不做”的方法。网站中一些高流量的地方可以预计算，而那些访问相对不频繁的页面可以是完全动态的。除了 Amazon，一般都是零售商决定了你在主页上能看到什么内容，每个人从主页上看到的都是相同的产品。网站可能叫出你的名字来欢迎你，还可能显示一个保存好了的购物车，因此可能有 100 左右的字节是客户特定的数据。而剩下的 100KB 对于每个客户都是相同的。

请记住

- 预计算改动不频繁的内容

任何在改变前你都展示多次的内容都可以预计算，以节省请求处理时间。将生成内容的时间

从每个请求中摘出来，放到部署过程中去。

10.4 调整垃圾回收器



对于 Java 应用，调整垃圾回收是提高性能最快捷和最简单的方法。以产品级容量和流量运行的应用如果未经调整，可能要花费 10%的时间来回收垃圾。这应该被减少到 2%以下。

现代垃圾回收器针对对象生存周期的典型双峰分布做了优化。大部分对象都是临时的，它们创建又销毁只是在毫秒间，Sun 将此称为早期失效（infant mortality）。更少数的对象是长时间存在的，经常就是程序执行的整个生存周期（单件类，服务注册表，服务器套接字，等等）。对象从“伊甸园”（eden space）中分配。如果躲过了一次垃圾处理，它们将被移入到“幸存者区”（survivor space），“伊甸园”和“幸存者区”都是“新生代”（young generation）。

当垃圾回收器运行时，它采取的第一步操作就是检查“伊甸园”中的活动对象。如果没有活动对象，整个“伊甸园”就会被很快回收。其中的任何活动对象都将被移动到一个“幸存者区”中。因此对于垃圾回收器而言，快速地舍弃对象并不是问题。不同的垃圾回收阶段将最终把幸存者移动到“老年代”（tenured generation），后者被垃圾回收器检查的频率远低于“新生代”（第三代是永久代——permanent generation，存放类和方法定义）。

通过在 JVM 启动阶段设置-verbosegc 参数，你可以将垃圾回收器的行为可视化。垃圾回收器的报告在控制台输出，所以你需要确定标准输出被定向到某处（对于应用服务器，这通常需要修改启动脚本）。如果使用的是 Java 5 以上的版本，你可以使用 Java SDK 自带的 jconsole 工具。图 10-1 显示了一些 jconsole 报告给你的内容。内存页显示了堆使用情况，以及花费在垃圾回收上的时间。这是一个相对空闲的日期服务器，显然没有承受过度的垃圾回收。

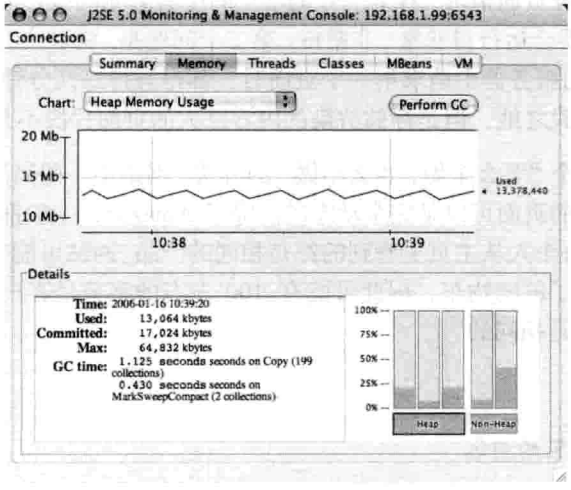


图 10-1 jconsole 内存页

一旦你看清了垃圾回收的规律，调整垃圾回收器基本上就是确保足够的堆大小，和调整控制各个代相对大小的比率。

Sun 在网上有关于垃圾回收器调整的很好的指南。[1]而且，通过调整垃圾回收器的练习，你会经常注意到内存泄漏的问题。

垃圾回收器调整既是科学也是门艺术。垃圾回收器的行为完全来自应用程序的行为和设定的模式。每次代码发布都会在某些方面改变环境，甚至相对小的代码发布也能引入用户新的行为：升级、易用性变化、突出昂贵的功能，等等。对于某次发布所做的完美无缺的调整（如果真能做到的话），对下一次发布则可能是完全错误的。我在第 17 章讨论透明性时会说到，你需要一个常规流程来反复地调整设置。

小乔爱问……

对象池怎么样?

在Java的早期版本（1.2左右），长生存周期对象的做法很流行。我清楚地记得曾被告知“在Java中第二贵时的事情就是创建一个对象（第一位的是创建新的线程）”。解决办法显然就是任何时候都要尽量避免创建新对象，而应该持有对象，并复用它们。

这是否正确，一直是Java社区辩论的热点话题之一。不管怎样，对于当前的JVM，对象池并不是答案。为了避免创建新对象，一些系统设计得有些荒唐。它们添加了太多复杂度和记账，葬送了获得的一切性能改进。

下面的表格显示了一个简单的应用程序的时间，它使用NameFormatter来格式化50 000个名字。在使用对象池的配置中，使用了Jakarta的commons-pool包建立对象池来复用NameFormatter。在不使用对象池的配置中，创建了50 000个独立的NameFormatter。数据清晰地显示出，过度的记账开销甚至都超过了创建对象的开销（所有的测试都是在JDK 1.4.2下做的）。

OS	CPU速度	开销（使用对象池）	开销（一次性创建）
Windows XP Pro	1.86GHZ	20.30%	10.17%
Linux 2.6.14	2.66GHZ	31.46%	23.42%
Mac OS 10.4.4	1.67GHZ	24.69%	15.69%

只有在创建对象真正贵时的才使用对象池，比如网络连接、数据库连接和工作线程。

① 对 JDK 1.4.2，参见 http://java.sun.com/docs/hotspot/gc1.4.2/index.html。对于 Java 5，参见 http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html。

请记住

- 在生产环境下要调整垃圾回收器

用户的访问模式差异很大，最优设置相去甚远，因此你不能在开发或者测试时调准垃圾回收器。

- 继续下去

你将需要在每一次重大程序发布之后调整垃圾回收器。如果你有一个年度需求周期，由于用户流量在各功能间都有变化，你还需要在每年的不同时间进行调整。

- 不要把普通对象放到对象池中

唯一值得放到对象池中的对象是外部连接和线程。其他的一切，都交给垃圾回收器吧。

10.5 总结

资源池可能会产生吞噬容量的竞争。然而，如果做得好，它们也能大大提升容量。连接池每次最多可以使事务减少 500ms。连接管理有几种策略可用，包括“按页”、“按段”和混合方法，每种都各有优缺点。

缓存是一把双刃剑。使用得当，它将极大地提高系统容量。往往在应用中，缓存的空间可能增长到很高，缓存项从没有被换出，并且还缓存了廉价对象，得不到任何好处。限制缓存大小是至关重要的。Java 开发者应该使用软引用对象，在管理缓存对象时和垃圾回收器协同工作。

留意乘法效应来找到正确的提升方法。如果内容每周只变动一次，那么每天动态渲染内容一百万次会把你的系统推到乘法效应的最糟糕的一端。好处每周只得到一次，代价却是每天要承受一百万次。在底层内容变动时渲染内容一次，并存储预计算结果，你得到的好处就是每天节省一百万次的计算，付出的代价只是每周管理一次缓存。

垃圾回收器调整对于 Java 应用程序非常重要。这不是一次性的工作，而是一个持续不断的流程。每次代码发布都可能改变垃圾回收模式，而需要你做一次调整。

Part 3 第三部分

一般设计问题

本部分内容

- 第 11 章　网络连接
- 第 12 章　安全
- 第 13 章　可用性
- 第 14 章　管理
- 第 15 章　设计总结

第 11 章
网络连接

数据中心的网络连接早已不是应用程序级别的套接字 API 了。数据中心的网络设计比桌面电脑网络更加需要冗余、安全和灵活。在这个环境中的应用程序要运行良好，我们需要做更多的工作。

11.1 多宿主服务器

数据中心的机器与开发或测试环境的机器最大的不同在于多宿主。几乎数据中心的每台服务器都是多宿主的。多于 1 个 IP 地址的服务器叫作多宿主服务器，它同时存在于多个网络中。这个架构通过在高度安全网络中分隔管理和监视来提高安全性。它通过把大流量（比如备份）与产品流量分隔开来提高性能。你一定遇到过由于备份而网络减慢的状况。备份能让任何网络饱和，因为它们总是以全负荷运行。如果你的网络更快，也就能让备份在一定程度上完成更快。这些网络有不同的安全需求，而没有意识到多网络接口的应用程序会很轻易地接受来自错误网络的连接。例如，它可能接受来自产品网络的管理连接，或者对备份网络提供产品功能。

如图 11-1 中，单个服务器有四个不同的网络接口。在 Linux 中，这些会是 `eth0` 到 `eth3`。

对于 Solaris，它们可能是 `ce0` 到 `ce3` 或者 `qfe0` 到 `qfe3`，这根据网卡和驱动版本的不同而不同。Windows 默认情况下会给这些接口又长又怪的名字。

在这四个接口中，有两个是专门留给“产品”流量的，它们用来处理应用程序的功能。如果是 Web 服务器，那么它们处理输入请求然后发回响应。在这个例子中，两个接口都用于产品流量。因为它们运行于不同的交换机，服务器似乎为高可用性而配置。这两个接口可能是负载均衡的，或者它们可能作为失效备援对。两个不同的 IP 地址的数据包都到达这个服务器，这意味着对两个地址很可能都有 DNS 项。换句话说，这台机器的名字不止一个！它有内部主机名，这是由 `hostname` 命令返回的，但从外部，可以经由多个名字访问到主机。

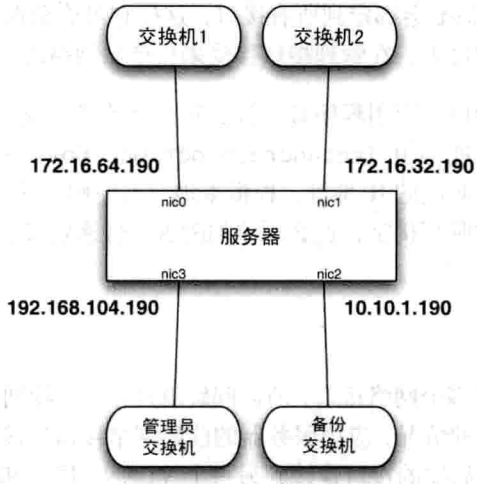


图 11-1 多网络接口

另一个多个产品接口的常见配置是接口绑定（bonding 或 teaming）。在这个配置中，两个接口共享一个常用的 IP 地址。操作系统确保一个数据包只经过一个接口。绑定的接口能被配置成自动均衡输出流量或者优先选择一个或另一个链接。连接不同交换机的绑定接口需要在交换机上进行一些额外的配置，不然的话能产生其他的路由环路。如果你在数据中心造成路由环路，那你就出名了，这当然不是什么好事。

两个其他的“后端”接口用于特殊用途的流量。因为备份突然传输大量数据，它们可能让产品网络阻塞。因此，数据中心好的网络设计是把备份流量分隔到自己单独的网段上。这有时候由单独的交换机处理，有时候在产品交换机上由各个 VLAN 来实现。由于备份流量隔离于产品网络，应用程序用户就不会在备份时遭遇阻塞了。（如果服务器没有足够的 I/O 带宽来同时处理备份和程序流量，他们还是可能会遭遇阻塞。然而，此时其他应用程序的用户却不会了。）

最后，很多数据中心对于管理性的访问有特定的网络。这是重要的安全保护，因为诸如 SSH 的服务可能被绑定到管理接口，因此不能被产品网络访问。这在防火墙被攻击者攻破，或者服务器处理一个内部的应用而且在它前面没有部署防火墙的时候，就很有用。

多数接口影响了应用程序软件。默认情况下，监听套接字的程序会在任何接口监听连接尝试。例如，Java 的 ServerSocket 类在 Java 5 中有 4 个构造函数。其中三个绑定到服务器的任何接口上。只有那个较长的构造函数能指定它需要绑定的具体本地地址。

```
InetAddress addr = InetAddress.getByName("alpha.example.com");
ServerSocket socket = new ServerSocket(80, 50, addr);

Socket local = socket.accept();
...
```

没有地址，ServerSocket 会绑定到所有接口，这样它可能会在产品服务器接收来自备份或者管理接口的报文，或者倒过来，在管理接口接受来自产品网络的报文。

为了决定要绑定哪个接口，应用程序必须被告知自己的名字或 IP 地址。这很不同于多宿主服务器。开发中，服务器总能调用 InetAddress.getLocalHost()，但是在多宿主机器上，这只能返回服务器内部主机名对应的 IP 地址。根据本地命名原则，这可能是任何一个接口。因此，服务器应用程序如果需要监听套接字，就必须添加配置属性来定义服务器应该绑定哪个接口。

11.2 路由

因为产品服务器通常有多个网络接口，所以问题就产生了，特别类型的流量应该穿过哪个接口呢？例如，比较常见的一种情况，应用服务器的前端网络接口连接到一个 VLAN，用来和网络服务器通信，同时它有一个后端的接口连接到另一个 VLAN，用来和数据库服务器通信。这种情况下，服务器必须被告知，为了到达特别的目的 IP 地址，应该使用哪个接口。

如果服务器临近，路由会比较容易，它们只基于子网地址。这个例子中，应用服务器的后端接口很可能与数据库服务器共享一个子网，同时前端接口很可能和 Web 服务器共享一个子网。但如果是远程服务，有可能包含第三方服务，这时路由就有点麻烦了。

考虑一下第三方的 spam cannon 服务。由应用服务器到 spam cannon 的数据很可能不通过公共的因特网。它很可能通过一个 VPN，路由经过应用服务器的后端接口，然后到达这个服务。

spam cannon：从事大容量电子邮件传输的供应商，通常基于名称和地址的 XML 摘要。他们用订阅者收集的个人数据填充在漂亮的、预定义的 HTML 模板中。他们有大带宽连接，使用算法来避免破坏小型 ISP 的 SMTP 服务器。通常这些工作成果直接进入了“垃圾邮件”夹。

要想正确解决这些路由问题需要关注每个集成点。如果路由出错可能降低系统可用性，更糟的是，会暴露客户数据。对远程系统的每个连接，我推荐都将记录保存在电子数据表或微软的 Access 数据库中，数据包括目的主机名、地址和期待的路由。某天，某人在写防火墙规则时会需要这些信息。

11.3 虚拟 IP 地址

不管相信与否，并非每个应用程序都适合在集群运行。有各种原因，使得这些应用程序一次只能在一台服务器上运行。没有在多服务器上冗余的运行，你怎么可能获得高可用性呢？

集群服务器就是答案。集群服务器扮演了类似于其他程序的控制器的角色。比如 HP

ServicGuard、Veritas 的 Cluster Server 或微软的 Cluster Server[①]这样的集群服务器，就运行在多台服务器机器上，让这些机器协作，以确保一个具体的“包”只在集群的某处运行一次。通过为应用程序、文件系统和 IP 地址配置包，有可能让集群服务器协调完成一台服务器干净的关闭和另一台服务器上的启动。大多数时候，应用程序本身不知道它已经在集群中运行了。

假设在一个服务器上面运行的某个关键应用失效了，但这个应用不是专门为集群写的。失效备援节点的集群服务器会注意到从失效的服务器缺乏常规的心跳包。这个集群服务器就会判断该原始服务器已经出故障了。

它会在副服务器上启动应用程序，包括装载任何需要的文件系统。它也会接管分配给集群网络接口的虚拟 IP 地址。

两种虚拟 IP 地址

不幸的是，“虚拟 IP”这个词被过度使用了。通常，它是指一个 IP 地址没有严格绑定到一个以太网 MAC 地址上。集群服务器用它来迁移集群成员地址的所有权。负载均衡器使用虚拟 IP 来使很多的服务（每个都有自己的 IP 地址）映射到少量的物理接口上。这里有一些交迭，因为负载均衡器成对出现，所以作为“服务地址”的虚拟 IP 也能作为“迁移地址”的虚拟 IP。

虚拟 IP 地址是能按需从一个网卡转移到另一个需要的网卡上的 IP 地址。在任一时间，只有一台服务器拥有这个 IP 地址。当地址需要被转移时，集群服务器和操作系统分工协作，在 TCP/IP 协议栈的底层做一些工作。它们将 IP 地址对应到新的 MAC 地址（硬件地址），然后广播这个新的路由（ARP）。图 11-2 描述了活跃节点失效前后虚拟 IP 地址的状况。



图 11-2 虚拟 IP 地址迁移

① 后来更名为微软 Windows Server 2003 Clustering Services（集群服务）。这名字真够绕的。

虚拟 IP 地址通常被用作主动/被动数据库集群。客户端只受命连接到虚拟 IP 地址的 DNS 域名，而不是连接到集群中任何节点的主机名。这样，不管哪个节点使用这个 IP 地址，客户端都能连接到相同的域名。

当然，这种方法不能迁移应用程序的内存中的状态。结果，任何关于交互的非永久状态都会丢失。对数据库而言，这包括没有提交的事务。一些数据库驱动，比如 Oracle 的 JDBC 和 ODBC 驱动，会自动地重新执行由于故障而终止的查询。更新、插入或存储过程调用不能被自动重复执行。因此，应用程序通过虚拟 IP 地址的数据库调用都要在失效发生时做好准备处理 `SQLException` 异常。

通常，如果你的应用程序通过虚拟 IP 调用了任何服务，它必须处理好这种可能，下一个 TCP 包不会和上一个包到达同一个接口。这可能在奇怪的地方产生 `IOException` 异常。应用程序的逻辑必须准备好处理这种错误，要和“destination unreachable”（无法达到目的地）的错误区别处理。如果可能，应用程序应该重新尝试对新节点的请求（但先看看 5.2 节“断路器”讲到的一些关于重新尝试的重要安全限制）。

第12章 安　全

本书无意于做出关于应用程序安全性的完整描述。本章谈论的安全话题只涉及软件架构、操作和运营的交叉部分。

12.1 最少特权原则

“最少特权”原则要求进程应该具有完成任务所需的最少权限。对于应用软件，绝不应以 root（UNIX/Linux）或 Administrator（Windows）身份运行。这些应用程序做所有事都用非管理员用户身份。

我曾见过一台 Windows 服务器被以 Administrator 登录了好几个星期，是采用远程桌面登录的，因为那是某些软件需要的。（这个特别的软件包也不能以 NT 服务运行，所以最终只是一个运行了很长时间的 Windows 桌面程序。这绝不是我说的数据中心时刻准备着！）

需要以 root 身份执行的软件自动成为骇客们的目标。用 root 身份运行的软件的缺陷自然而然成为了严重的问题。一旦骇客获得了 root 访问权限，唯一保证服务器安全的就只能是重新格式化和重新安装。更糟糕的是，对于水平可扩展的应用，你可能不得不对整个机群进行重装。

为了进一步隔离漏洞，每个重要的应用程序应该有它自己的用户。比如“apache”用户不该访问“websphere”用户。

打开端口号小于 1024 的套接字会是 UNIX 程序可能需要获得 root 权限的唯一原因。Web 服务器默认状态下通常需要打开 80 端口。负载均衡器（参见 13.3 节）后面的任何 Web 服务器能够使用任何端口，包括 1024 号或更大的端口号。只有负载均衡器需要监听 80 端口。

不受负载均衡器管理的 Web 服务器（假设这样做有充足的理由，但我一时还想不出）为了监听 80 端口，就需要以 root 权限运行。然而，看看 Apache，你会发现它使用“权限分离”，在打开套接字以后故意放弃了 root 权限。它使用低级别的 C 函数来把权限降级到配置的用户（通

常是“apache”)。而且这是单向的，进程一旦放弃，就不能重新获得 root 权限了。

12.2 配置的密码

密码是应用程序安全的阿喀琉斯之踵。很明显，不可能在应用服务器每次启动的时候，都由人来输入密码。因此，数据库密码以及和其他系统进行鉴权用到的证书必须配置在永久文件中。

只要密码保存在文本文件中，它就是脆弱的。任何授权访问客户信息数据库的密码对于攻击者来说都价值连城，而一旦泄密或者遭到黑客勒索，公司都会损失惨重。这些密码需要以最高的安全级别保护起来。

至少，产品数据库的密码应该和其他配置文件分开保存，尤其不该保存在软件安装目录。(我曾经见到运营部门将整个安装文件夹打包，然后发回给开发部门分析。在人们怀疑产品配置出现问题或软件未正确安装和配置时，就经常会这么干。)包含密码的文件应该只对它的拥有者可读，也就是应用程序的用户。如果应用程序以一种能执行权限分离的语言编写，那么在应用程序降级权限之前读出密码文件就是合理的。这时，密码文件被 root 拥有。当心！如果应用程序将这些文件内容保存在内存中，那么内存转储中也会包含密码。对于 UNIX 系统，核心文件只是应用程序内存转储。如果攻击者能触发核心转储，访问到服务器的文件系统，那么他就能获得密码。最好在产品程序中禁止核心转储。对于 Windows 系统，“蓝屏”就表示内核错误，会同时进行内存转储。转储文件能用微软内核调试工具分析，取决于服务器的配置，它有可能包含整个机器物理内存的拷贝，其中就有密码。[①]

密码隐藏在加密文件中保存密码，这能将安全问题降级为单个加密密钥的安全，而非多文本文件安全。这能够辅助保障密码，但其本身不是一个完整的解决方案。因为容易在不经意间改变或覆盖文件权限，所以应该使用诸如 Tripwire[②]或开源版 Tripwire[③]等入侵检测软件来监视这些重要文件的权限。

① 事实上，有个注册表的编辑项能启用一个组合键(我这里不打算告诉是哪个)，它能迫使内存转储和系统停止。如果你看到“蓝屏死机”，并且有条文本“目标用户手动产生故障转储”，那么说明你受到攻击了。实际上，最好是确保这项“功能”在产品服务器上不被启用。

② 参见 http://www.tripwire.com/。

③ 参见 http://www.sourceforge.net/projects/tripwire。

第13章
可用性

不顾成本谈需求会助长脱离现实的欲望。问一群孩子他们想要多少冰淇淋，答案通常会是“所有”。吃一加仑冰淇淋的各种成本，孩子们没有概念：直接成本是买冰淇淋花的钱，还有间接成本如危害健康、长胖、反胃、肚子难受。

这一章讨论如何平衡高可用性和降低成本，这绝不容易做到。它们是一对矛盾，要保证可靠性就不可避免地要提高成本。

13.1 收集可用性需求

询问你系统的发起人，“需要多高的可用性？”，你可能得到以下两个答案。经验不足的发起人会不假思索地回答“100%”；有常识一些的会说“5个9”，因为这听起来更酷更有技术含量些。但是真的需要5个9吗？

做可用性决策的合适方法是直接从财务角度看：真实的成本和可避免的损失。例如，把“98%的可用性”转换成每个月864分钟的停机时间。停机会带来直接损失，那就是损失的收入。假设网站在每天高峰时间带来每小时1500美元——高峰时间是最可能停机的时候，那么98%的可用性的最大损失就是21 600美元。现在比较一个更可靠的系统的停机时间。提高可用性到99.99%会减少停机的成本到每月只有108美元，相对98%的可靠性获得了21 492美元。

所以，系统就应该构建到99.99%的可用性对吗？这就如同前面那个冰淇淋的问题。不考虑成本，欲望是无止境的。可用性的每个“9”都会提高10%的开发成本，和每年20%的运营成本。在这个例子中，会增加的生存期成本，也就是5年生命期的开发和运营成本之和达到98 700美元。花98 700美元来节省1 289 520美元似乎是一个合理的财务选择。

可用性	98%	99.99%
每月最少宕机时间	864	4
每月宕机成本	$21 600	$108
新的成本	$0	$98 700
净节省	$0	$1 289 520

13.2 记录可用性需求

成心想造成讨厌的冲突吗？选个有多个含义的模糊的词，强迫人们达成协议，投上大量的资金，只要一两年后你就能看到他们在打架了。

当然，我说的是定义不清的可用性服务水平协议（service-level agreement，SLA）造成的后果。SLA 定义就如同你的医疗保险计划的细节一样，没有人会仔细地看，除非有坏事发生。如果系统启动后的一年，你们又在讨论 SLA 的定义，那很可能是在事故之后的火爆讨论。一旦发生这种情况，就悔之晚矣。

应该在一开始就和系统发起人协商好精确的 SLA，这样就能避免（至少是减少）事后的相互指责。只是白纸黑字地写上“这个系统应有 99.9%的可用时间”是不够的，这句话的每个单词都闪烁着模糊的光芒。

什么是“这个系统”？“这个系统”很可能包含对于企业内外其他系统的调用。你要考虑所有这些情况吗？我不会！而且，如果系统使用了断路器这样的稳定模式（参见 5.2 节），在某个时候，系统整体可能保持着运行，能响应请求或生成网页，但是有的功能可能就不可用。所以，在 SLA 中不要声明“这个系统”可用，最好是说明系统具体的哪些功能可用。例如，连锁酒店可能关心其网站的几个关键功能要可用，比如查找酒店位置、在线订房、加入会员、活动预定等。每个功能都有不同的重要性，活动预定和在线订房都会产生收入，所以它们需要最高的 SLA。当心 SLA 倒置反模式（参见 4.10 节），你不可能提供的 SLA 比功能外依赖的最差 SLA 还高。例如，连锁酒店的会员服务通常由第三方处理，因此这个 SLA 最好情况是第三方供应商的 SLA。

一旦细化到每个功能或商业过程，你就能定义需要什么样的可用性级别。“可用性”本身需要去定义，这一点要学学人家律师。这个功能该怎么检查？是人为去点点鼠标？（我希望不是。）还是让一大帮用户把问题报告给帮助台？（我真的希望不是！）最好是有自动化的系统通过执行模拟事务来检查可用性。SLA 应该定义用什么设备来监视功能的可用性。再进一步，它应该定义监视设备要如何报告问题。

模拟事务：模拟真正的用户提交给系统的工作。你必须有一些办法来表示这是用于监测的，以避免污染产生的数据——例如使用指定的用户 ID。

如果一个功能起作用，但是需要 27.5 分钟来响应，那么大多数用户都不会认为它“可用”。

因此，这涉及了时间的因素。如果事务能在 50 毫秒内反应，但是返回的是错误呢？那也不是真正可用的，因此它也需要定义良好响应应该是怎样的。如果一个功能摇摆不定，但是在监视设备检查时似乎又正常（就像你车出了毛病，送去检修却查不出问题），这种情况呢？好的定义应该明确下面这些变量：

- 监视设备执行模拟事务多长时间一次？
- 事务每一步的最大可接受响应时间是多少？
- 什么样的响应码或文本模式表示成功？
- 什么样的响应码或文本模式表示失败？
- 模拟事务执行的频率如何？
- 来自多少个地方？
- 数据记录在哪？
- 计算可用性百分比的公式是什么？依据的是时间还是采样的数量？

当事故发生时，一纸协议固然很脆弱，但是有了这些定义就能将问题的重点放在数据上，而不是相互做人身攻击。

13.3 负载均衡

水平扩展系统通过做乘法来获得可用性和可扩展性。添加机器既提高了容量也提高了应对峰值冲击的弹性。水平扩展架构中使用的小服务器能让成本更低，并可以一点点地增加容量。有什么理由不喜欢呢？

峰值冲击：对系统短暂而巨大的冲击。就像用锤子猛砸一下。

水平自动扩展来构建系统也暗示了某种负载均衡。负载均衡是对服务器池或农场分发请求，以便以最短的可行时间正确处理请求。本书后面经常会谈及这些负载均衡方式的设计和场景。

轮叫DNS

轮叫 DNS（DNS round-robin）负载均衡是我这里讨论的最古老的技术，可以追溯到 Web 发展的早期。它作用在 OSI 协议栈的应用层（第 7 层），但并非作用在真实请求期间，而是在地址解析期间。

这个技术简单地将几个 IP 地址和服务名关联起来，通常用于中小型商业网站，因此它不是对“bobscleaners.example.com”找到单个 IP 地址，而是客户端会获得多个地址中的一个。每个 IP 地址都指向单个服务器。客户端因此连接到服务器池中的一个，如图 13-1 所示。

虽然这能达到对一组机器分发工作的目的，但按其他一些重要评判标准它却表现较差。首先，

池中的所有服务器必须可“路由”。也就是说，虽然它们在防火墙之后，但是它们的前端 IP 地址必须让客户端可见并可达。现在，这样做会招致攻击。

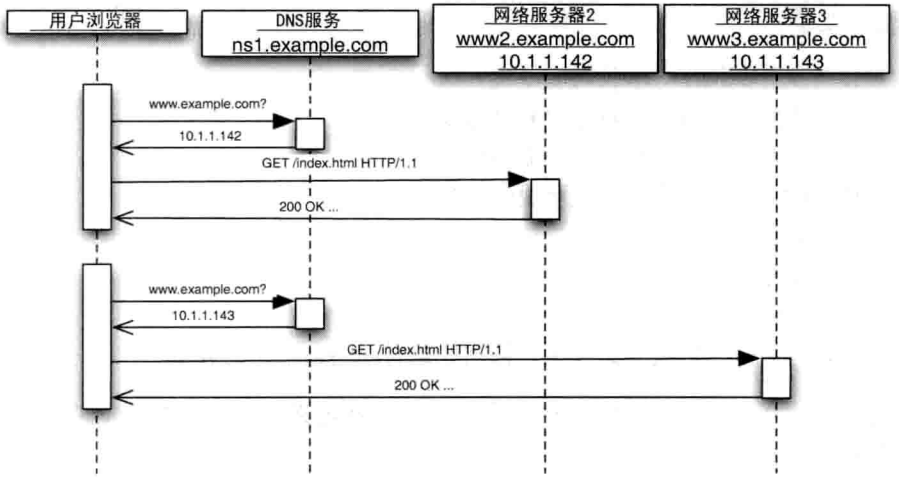


图 13-1 轮叫 DNS 负载均衡

除了安全考虑，轮叫 DNS 方法让太多的控制掌握在客户手中。因为客户端直接连接到一台服务器，因此在某台服务器停机时就没有机会重新定向了。DNS 服务器不知道网络服务器的健康状况，因此它会向停机的服务器继续发送 IP 地址。而且，用轮叫的方式发出 IP 地址不保证能负载均衡，只是初始连接均衡。有些客户端比其他的要消耗更多资源，这样就造成工作负载不平衡。这时，某台服务器变忙，DNS 服务器却无从知道，它只是将分到这台服务器的连接不停地发送到摇摇欲坠的它身上。

当调用系统又是一个长期运行的企业系统时，轮叫 DNS 的负载均衡方式是不适用的。由 Java 构建的任何内容都会缓存从 DNS 收到的第一个 IP 地址，这使得未来的每个连接都指向相同的主机，让负载均衡彻底没法实现。

一些 Apache 配置也执行轮叫方式的 URL 重写。用户能够看到，这时 www.example.com 突然变为 www7.example.com。不幸的是，这种方式比轮叫 DNS 表现更糟，因为用户倾向于将各个服务器而不是将“front door”地址标记为书签。

反向代理

反向代理服务器作为一个请求拦截器，能解决轮叫 DNS 方法的限制。这时，DNS 将服务名解析为精确的 IP 地址。如图 13-2 所示，监听 IP 地址的设备是典型的主机，配置使用“反向代理服务器”模式运行代理服务器。

正常的代理把多个输出调用通过多路传输发到单个源 IP 地址。反向代理服务器恰好相反，它将输入调用传输到单个 IP 地址，然后散发到多个地址。以网络服务器 Squid 为例，这是一个流行的开源代理服务器，它就是个很好的反向代理服务器。[①]Apache 的 `mod_proxy` 也允许特殊用途的服务器对其他的 Apache 服务器农场或集群扮演反向代理。

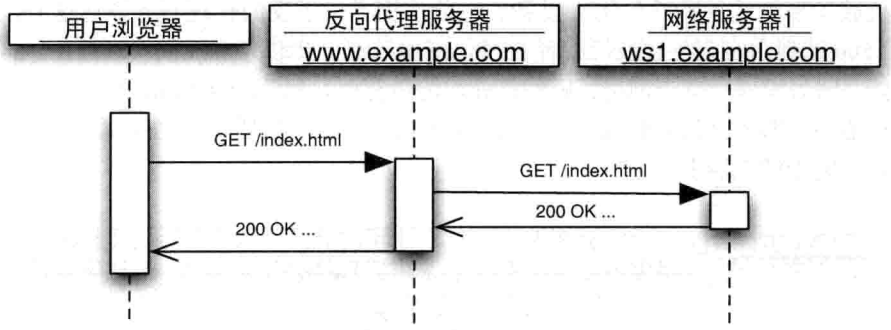


图 13-2 反向代理服务器

就像 DNS 轮叫一样，反向代理服务器在应用层做了一些取巧的事。它们不透明，但是要适应它们并不费力。应用服务器和网络服务器必须配置为对非自己的主机名产生 URL。正如 Apache 的“Common”日志格式，登录请求的源地址无用，因为它仅表现代理服务器。[②]

除了负载均衡，你能配置反向代理服务器缓存静态内容，这样能减轻网路服务器的负担。这对于内部网络，能减少流量。（从反向代理服务器发送静态内容，只要经过一两个链路到输出网络。如果来自网络服务器，它可能会跨越两三个。）如果网络服务器导致了系统的容量限制（参见 8.2 节），那么取消这个流量会提高系统的整体容量。当然，如果代理服务器本身造成了限制，那也就没什么帮助了。

世界上最大的反向代理服务器“集群”是 Akamai。Akamai 的基本服务像 Squid 一样精确作用，配置为缓存代理。Akamai 比 Squid 有优势，包含它有大量的与目标用户临近的服务器，但是它们逻辑上是相同的。[③]

因为反向代理服务器被使用在每次请求中，因此它的负担增长很快。一旦你开始不得不在反向代理服务器前端添加一层负载均衡，也就是需要查看其他选择的时候了。而且，因为代理服务器在每次请求的中间，它不得不跟踪哪个原始服务器是健康而且能响应的。不幸的是，最常使用

① 参见 http://www.squid-cache.org/的配置命令。

② 然而，有可能使用一个自定义的日志格式来记录 X-Forwarded-For 头，Akamai 和其他表现良好的代理都添加了这个头到请求中。请注意，行为不好或恶意代理不可能遵守这部分标准。结果，当你最需要跟踪流量源头时，比如一个攻击，这个头最不可靠。在这种情况下，你必须依靠相应的代理日志文件。

③ 请注意，我只是在说 Akamai 的基本服务，目前称为网络应用加速器。它的其他服务远远超过了 Squid!

的反向代理服务器 Squid 和 Apache 都没有这个功能。它们会将输入请求定向到宕机的服务器上，等待超时，然后给调用者返回错误。

硬件负载均衡

硬件负载均衡器是种特殊的网络设备，它的作用类似于反向代理服务器。这些设备，比如 Cisco 的 11500 系列内容服务交换机或者 F5 的 BigIP 产品，提供了和反向代理软件一样的拦截和重定向功能。因为操作接近于网络，硬件负载均衡器有更好的管理和冗余功能。例如，CSS 能够周期性检查池中的服务器健康状况，如图 13-3 所示。负载均衡器将死亡的服务器移除服务池，将连接定向到健康的机器上。

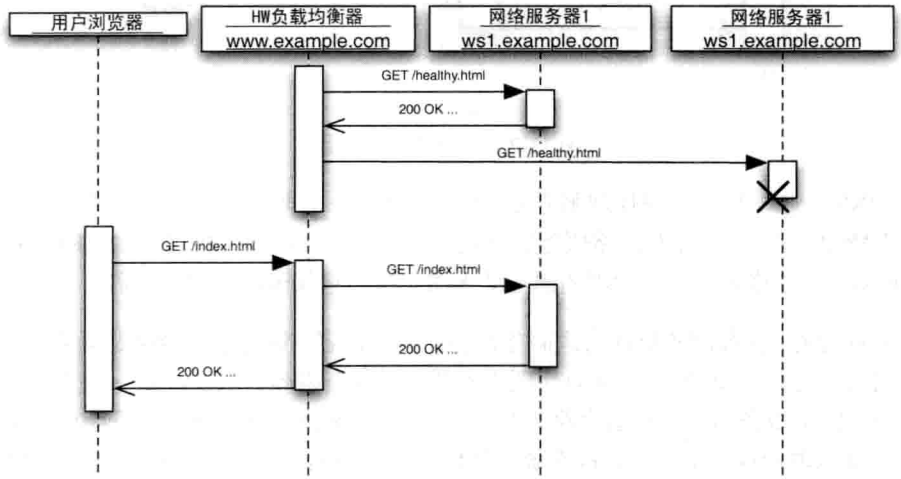


图 13-3 硬件负载均衡

Apache 2.2 的负载均衡

Apache 2.2 包含了新的 mod_proxy_balancer 模块。这是一个智能的功能更丰富的负载均衡模块。它能按比例地“均衡”网络服务器的负载，虽然你需要自己设置比例。

Apache 作用在古老的 OSI 模型的第 7 层，[①]它是一个应用程序。专用的硬件会更快并达到更高的容量。但是对于很多网站来说，省去一对 50 000 美元的硬件负载均衡器的成本比起降低上限来说更重要。此外，没有设计是静态的，因此如果网站使用开源的负载均衡器限制增高时，它们总是用高性能的硬件解决方案代替。

① 参见 http://en.wikipedia.org/wiki/OSI_model。

硬件负载均衡器是跟应用程序有关的，并提供在 OSI 协议栈第 4 到 7 层切换的能力。实际中，这意味着它们能对任何面向连接的协议负载均衡，而不仅仅是 HTTP 或 FTP。我曾见到过用这种方式成功地对一组搜索服务器实现负载均衡，这些服务器并没有自己的负载管理器。它们也能将流量从一个实体网站转移到另一个，这对在灾难恢复时将流量转到失效备援网站上非常有用。

SSL 对这些解决方案提出了挑战。硬件负载均衡器通常被用作 SSL “加速器”。坦白地说，在我看来，这让负载均衡器站到了摩尔定律和可扩展性的对立面，摩尔定律说网络服务器每 18 个月都能替换成更快的机器。毕竟，比起硬件负载均衡器来说，会有更多的网络服务器。而且，网络服务器很可能没干多少活，然后负载均衡器在每次请求中都被用到。使用负载均衡器来处理 SSL 解密能让 SSL 证书管理更容易，因为只要更新一对设备，而不用更新十多台网站服务器。实际上，如果负载均衡器对不同的域处理多个 VIP，那么你的所有 SSL 证书都能放在相同的一对机器上（假设它们足够大来处理负载！）。

SSL：安全套接字层。为保密和完整性而进行了加密的连接。相对来说是 CPU 密集型的。

硬件负载均衡器能提供很多应用级别的服务，包括 HTTP 错误码检查和基于网站服务器响应的不同出错页重定向。对于 HTTPS，你有两种选择：或者为 SSL 使用负载均衡器并保存应用级别的服务，或者在网站服务器终止 SSL 连接，而用负载均衡器来做智能切换并放弃应用级别的服务。如果尝试直接传递 SSL 连接到网站服务器，同时也检查内容的话，负载均衡器会在浏览器和服务器之间对加密渠道扮演中间人攻击。

当然，这些机制最大的缺点是成本。低端配置需要支付 5 位数；高端配置，比如那些需要第一层零售商的配置，很容易开支到 6 位数。

13.4 集群

负载均衡并不需要不同服务器间的合作。当服务器意识到彼此，然后主动参与负载分发，它们就形成了一个集群。集群能用来负载均衡，比如主动/主动集群。它们也能用来在失效的时候做冗余，这称为主动/被动集群，意思是一台服务器处理所有的负载，除非它失效了，那么被动服务器接管它的工作并变为主动。

全负载均衡的服务器农场接近线性扩展。负载均衡集群却不是。集群使用一些心跳通信和状态同步。这种开销的结果是，集群容量低于线性扩展，并可能随着服务器数量的增多而衰减剧烈。这限制了集群的规模，虽然这种限制因服务器而异。

心跳：集群的服务器之间发送的数据包，通常在不同的网络之间。表示“我还活着”。

一些应用服务器构建在它们自己的集群中。WebSphere，WebLogic，Oracle 和微软的 SQL Server 都有自己的集群构建技术。伴随的应用软件会配合其可用性和失效备援。这些应用程序不

可避免地有一定的“主控制”节点来负责将工作定向到集群的不同节点上。这些控制节点可能成为架构中的弱点（比如，如果只有一个），并通常将会首先成为容量的限制。

对于没有自身本地集群的应用程序，有可能在集群服务器的控制下运行，比如 Veritas Cluster Server 或微软 Windows Server Clustering Services。这些集群服务器扮演了外骨骼的角色，作为子进程运行集群应用程序。集群服务器彼此传递心跳包，并通常在网络驱动上使用一个“法定数”来同步它们的活动。如果集群服务器探测到一个节点失效了，它们会执行一系列预定活动来让集群应用程序在存活节点上备份。这些活动包括接管一个或多个文件系统，从头启动应用程序或者从内存镜像中激活一个应用程序，然后接管虚拟 IP 地址。

我对集群服务器既爱又恨，它们又非凡又笨拙。它们能对未经特别设计的应用程序添加冗余和失效备援。但是配置集群服务器本身极其考究，并且应用程序通常在失效备援过程中会遇到一些小故障。最大的缺点是这些很可能发生在主动/被动模式中。因此，冗余能达到，但是可用性却没达到。我将集群服务器看作是应用程序的创可贴。

第14章 管　理

如果你的系统让管理员很易用，它就能运行良好。而且，你会发现它很容易获得帮助和操作资源。相反，如果你的系统让管理员难用或烦恼，它就会被忽略、藐视，无法正确实现，甚至还可能遭到破坏。

管理员是个艰难、不讨好的工作。在系统设计和架构时，从来没有人去咨询他们。他们总是在最后得到一些不成熟的软件，而且必须将它们融入到常规操作中去。

还有个更加基本的矛盾。开发者和用户都积极地看待改变。每次新的发布都意味着新的功能，对应有收入的增加、错误的修正和设计的改进。发布是增值的。对于管理员来说却截然相反。他们不得不为每次部署新版本干更多的活。而且，每次发布以后，系统会和以前表现不同。旧的日志消息或命令不见了，取而代之的是新的东西。旧的失效模式可能被改正了，但是却可能产生新的，没人知道如何检查出来。当每件事都学起来很艰难时，人就会变得保守。其实这两个观点都正确!

管理员既可能是强大的同盟，也可能是强悍的敌人。(毕竟，和你相比他们和IT经理谈得更多。)好消息是并不需要贿赂或宣誓来赢得他们的忠心。只需要理解他们的动机，然后尽力使他们工作更容易。他们工作容易了，就能做得更好，因为他们能够而且愿意去做。

下面这些方法可以让软件对管理员更友好，从而使他们工作愉快。

14.1 “测试和产品匹配吗？”

在部署失败时，或者一些错误就存在于产品表面时，人们不禁会问为什么这个问题没有在测试的时候发现。这个问题的问和答很重要。90%的时候，这个问题的直接推论是：“测试和产品环境的配置有差别吗？”

这个问题问起来容易，但是回答起来却很费劲，而且肯定你将会发现某个地方有所不同，最

简单的情况是主机名和IP地址不同。你可能在想“为什么答起来难？配置管理系统不都一样吗？”也对也不对。配置管理系统可以告诉你什么时候做出的改变，谁做的，有时还告诉你为什么。有些好的运营模块能够询问系统并找出差异。昂贵的部分是确定哪些差异是已知的、预期的、无害的，而哪些是意外的和有风险的。例如，在测试环境使用小规模的硬件很常见。一些供应商，特别是那些比如惠普之类的软/硬件提供商，使用相同的补丁包机制来提供固件升级、设备驱动补丁、操作系统功能升级和安全补丁升级。在这种情况下，不同的硬件模型可能需要不同的补丁集。

另一个层面，你也可以说，测试必须和产品不同。毕竟，如果使用相同的主机名和端口号，那就不叫测试了，那就叫产品！

以我的经验，失效的最常见原因并非配置。我曾花数小时来琢磨配置文件，调节预期的差异，标记没想到的差异。可是一段时间以后，当问题暴露时，才发现配置差异通常不是罪魁祸首。

大多数时候，真正的罪魁祸首是测试和产品上拓扑结构的不匹配。拓扑结构是什么？它是服务器和应用程序的连接和数量。如果你把每个服务器和应用程序实例都看成一个节点，而每个连接或依赖是一个弧，你就能画出系统的拓扑图。这可以用 UML 部署图画，虽然未免大材小用。

显然，让测试拓扑和产品相匹配的主要障碍是成本。下面这些方法可以让产品拓扑和测试相匹配，同时也不突破预算。

使它们分离

让它们分离。通常，应用程序在测试时共享主机，而在产品环境单独运行。这可能导致隐藏的依赖关系：两个应用程序可能希望一个目录同步内容。它们只会在测试时工作良好，因为是同一个目录，然后在产品环境并没有保持内容一致的机制。（最后一刻才运行 `rsync` 的 `cron` 作业不算！）假设你不能为测试买和产品环境相同数量的服务器，你该如何避免这样的隐藏依赖呢？

这种情况，我推荐用 VMware。VMware 可在一台物理机器上创建多台虚拟主机。每个虚拟机器的行为都像一台独立的服务器，它们有自己的操作系统、IP 地址和主机名。如果两个应用程序在产品环境运行在不同的主机上，但是在测试时又需要共享相同的硬件，那你就能让它们在共享的机器下运行于不同的 VMware 实例中。（这还能让你保存应用程序特别版本的快照，这在测试过程中非常有用！）

零、一、多

老话说得好，计算机科学中最有用的数字就是 0、1 和多。在 0 到 1 和 1 到多之间有本质的区别。有时候，你会在测试环境发现单个实例，而在产品环境有多个处于集群或服务器农场的实例。这可能让点对点缓存失效策略和多播或广播策略不同。这又是一个诸如 VMware 服务器虚拟化能帮助你匹配产品环境多样性的例子。如果在产品中运行十几个实例，你不一定需要在测试中

也运行十几个（虽然我后面会有相关讨论），但是你应该运行不止一个实例。

要买必要的设施

我曾见过防火墙和负载均衡器导致的数小时的停机，因为测试环境中没有这些。停机的成本超过了网络设备的成本。测试可能不需要顶级的模型，但是需要有同一个供应商和产品线的产品。毕竟，你会在哪里测试防火墙配置变化呢？

按你练习的方式表演

真正的比赛中，运动队会用训练时的战术打比赛。同样，如果开发团队从一开始架构就使用防火墙，团队也会做相应的设计。否则，系统 95%都完成了，却还要去殚精竭虑地想象防火墙会有什么影响，那就太痛苦了。但是，如果开发团队一直在有防火墙的环境中，他们会记录已经有的规则。

跟踪集成点并同时跟踪防火墙规则（参见 4.1 节）。如果没有调用其他系统，为什么防火墙会有漏洞？那就是集成点。另外一个好处是：这可能也是最需要对产品环境做出改变的配置参数。

14.2 配置文件

Ruby on Rails 喜欢惯例甚于配置，但每个我曾见到的企业级软件都有许多配置文件，里面包括了主机名、端口号、文件系统位置、ID 号、魔术键、用户名、密码和中彩号码。（最后一个是我加的。）一旦这些属性有错，系统就会出现故障。即便系统看上去还能工作，它也可能患上了不为人知的重症！

配置文件通常命名隐晦，深埋在代码库的目录结构中，或者就是难以理解。有个属性叫作 `hostname`，它究竟是“我的主机名”，还是“授权调用者的名称”，还是“秋天冬至时我调用的主机”？属性文件遭遇隐藏链接和高复杂性，这是最可能引起操作员错误的两大因素。这些文件包含了整个企业最敏感的信息，除了没有 CEO 的工资信息，产品数据库密码等信息都包含在这个文件里了。因此，它们高度机密，配置其他部分的偶然错误不可能被开发者发现，这些开发人员了解并关心 `wsdlServer` 和 `uddlServer` 的区别。

我见过最常见的配置结构设计错误是将产品配置与其他基本信息混在一起。例如，使用 Spring 框架来 wire beans 很好，但是 Spring 希望我将所有的配置放到一个文件中。[1]那不仅包括

① 有些方法来模块化这些文件，但我认为这些方法是雪上加霜。

了在开发和产品之间应该改变的属性，而且还包括对象实例和关系的重要细节。结果，管理员必须手动编辑 5000 行的 XML 文件来更新数据库密码，而其余的 4999 行都是地雷，显然很可能改错。所有这些冠之以“依赖注入”，结果对应用程序非常重要。弄得不好，不小心改动了密码以外的东西，应用程序必然会出故障。类似这种出故障的机会不可胜数。

决不能让管理员打破应用程序中的对象关联。否则就太折磨人了。只要有可能，请将产品配置属性与应用程序的基本框架分离。它们应该放在单独的文件中，以防止管理员不小心修改。

同样重要的是，产品的配置文件不应该放在软件安装目录下。安装目录可能在下次更新时覆盖。你真想让管理员在每次更新后都回忆他做出的所有改动然后再重新配置吗？更何况，管理员通常是一次将每个服务器的安装树拷贝以跳过费时的安装过程。（例如，安装 BEA WebLogic 9 的第一步是解压整个 500MB 的文件。仅这一步就要在 2GHz 的 x86 比萨盒上跑 10 分钟。）备份和恢复也成了问题。从磁带恢复能轻易用旧的配置覆盖新的配置。



小乔爱问……

为什么不将配置文件保存在版本控制中？

很久以前，程序员们为自己编写了一个防止文件和修改丢失的方案：版本控制。如果配置文件对管理员来说是大问题，为什么他们不使用版本控制来进行管理呢？

有些人这样做了。在一些高级的软件公司，管理员使用诸如Subversion和cfengine的工具来管理和部署配置。越来越多的IT公司使用系统管理套件，比如HP OpenView、BladeLogic和Opsware来管理配置。

但是在很多时候，IT运营管理人员没有时间来实施和推出版本控制，或者他们不知道这个概念。有些知道版本控制，但是他们从来不曾想过将它用于系统配置文件。（我没说这是个好的理由。）

任何时候，我都坚决推荐使用版本控制来管理配置文件，但是有几个限制。

- 使用安全的储存库。这些文件有数据库密码！
- 将版本控制与更大的改变控制过程关联起来。你应该能看到配置为什么改，而不只是看到它发生了。
- 直接从储存库中自动化授权改变的部署。
- 进行自动的审计过程。文件有时候会在恢复服务的事件管理过程中改变。它们有时在后续版本控制中被更新。你需要能够在不自动覆盖它们的情形下找出变化。

作为替代方案，你可以购买和部署BMC推出的27个产品，来帮你完成这一切。

因为相同的应用程序可能运行在数台机器上，所以有可能配置属性中的某个子集在每台机器上相同，而其他部分有区别。将这些属性放在不同的地方，不要让人总来问“这些有什么不同？”。而且，我发现定期做做验证，确保水平扩展层的机器是同步的（它们应该同步），这很有好处。即便有再强大的变化管理过程，也应该遵循“信任，但要验证”的规则。

最后，配置属性是系统用户接口的一部分。它们是系统提供给其最容易被忽视的一方的接口，即给让它每天保持运行的人。因此，属性名应该足够清晰，以便用户（管理员）能在没有“非受迫性失误”的情况下完成工作。

一个有用的约定是将属性根据它们的功能而不是性质来命名。不因它只是主机名而称它为`hostname`。否则，就像命名一个变量为`integer`，只因为它是整数，或者命名它为`string`只因为它是字符串。这逻辑上是对的，但毫无用处。相反，可能应该命名这个属性为`authenticationServer`。管理员看到这个名字，会去寻找 LDAP 或活动目录主机。

最流行的 Java 应用服务器，对于配置管理也是大杂烩。WebLogic、WebSphere 和 JBoss 每个都提供了漂亮的 HTML GUI 来编辑配置。

WebSphere 和 WebLogic 在识别属于你的应用而非其他系统的属性上做了可信赖的工作。JBoss 鼓励稍微混合一点。所有这些都使配置彻底混合在应用服务器的安装目录里。WebSphere 至少有一个“配置存储”的概念，能够用来做属性的主拷贝，并在将应用程序从一个环境提升到另一个时作为部署源。尽管如此，它们没有单独的位置来存储应用程序特定的属性和系统级的属性。

.NET 开发人员有更好的方法，但只是当使用最新的 Visual Studio 版本来管理配置时才可行。当然，只有最勇敢的（也是最愚蠢的）开发者才会尝试在没有 Visual Studio 环境下开发.NET 程序。

14.3 启动和关闭

程序在开发人员启动时出错，开发人员能看到错误，杀掉程序，然后修复错误。如果主机在半夜重新启动，应用程序在启动时出现故障，除非应用程序本身报告错误，否则没有人会知道。如果已经有了监视系统（参见第 17 章“透明度”），就可以像在日志文件中记录错误一样简单了。当然，要让应用程序能报告错误，首先它需要知道它在启动时出错了。

在应用程序中构建干净的启动序列能确保组建正确的顺序启动，并且在程序开始工作前启动序列必须成功完成。例如，它可能会绑定套接字，但是在主开关翻转之前不应该接受任何连接。把它看成是大早上即将开门的商店吧，你不会因为有一个员工到了就让顾客蜂拥而入。相反，你要等到所有人都各就各位，并做好了为顾客服务的准备。

如果应用程序需要一个连接池，当程序启动时，至少某些连接应该被初始化。这是另一种快速失效的形式（参见 5.5 节“快速失效”），并提供了有用的自检。没有数据库，应用程序当然不能很好地处理交易了！因此，如果连接池因为不能创建任何连接而初始化失败，那么整个程序就处在失效状态了。

在启动完成前不要接受连接。

请注意，这和启动时失败导致的终止和退出非常不同。运行的程序能够被查询内部状态（参见第 17 章“透明度”），但是被暂停的程序就不能。

干净地关闭与启动时的准备同等重要。正如零售商店在还有顾客时不会锁门一样，应用程序不应该粗暴地终止。相反，每个应用程序需要一个将现有事务完成并不接受新事务的模式。一旦在处理的事务都完成了，应用程序就能够退出。但是请确保使用超时规则（参见 5.1 节“使用超时”），否则可能关闭永远不会完成。

14.4 管理接口

美妙的 Java GUI 非常棒。它们让软件看起来更“企业级”。不幸的是，它们是产品的噩梦。GUI 的主要问题是各种点击。我不能用脚本来完成这么一堆点击按钮。Java 的 GUI 会拖慢管理员在服务器上做的手动操作（这可能有许多！）。例如，我曾处理一个有特定关闭顺序的管理系统，整个关闭序列需要进行点击，再等待数分钟，这要在 6 台服务器上进行。猜一下这个干净的关闭序列需要多久观察一下？如果改变窗口只要一个小时，我不能花一半的时间等待 GUI 啊。

远程访问 Java GUI 也是一个挑战。管理员通常通过 SSH 来登录系统。使用 X 连接或 HTTP 连接来路由很是痛苦。

最终的结果是，Java 的 GUI 对于长期的产品运营来说是个可怕的接口。最好的接口是命令行。有了命令行，管理员能轻松搭建脚本、日志和自动操作来维护你的软件。退一大步讲，但是仍然可行的是纯 HTML 的管理 GUI。因为 Perl 和 Ruby 有丰富的 HTTP 客户端库，因此对 HTML 脚本管理接口写脚本并不太难。

第 15 章
设计总结

在项目喧嚣的开发过程中，很难关注到这些主题，特别是一旦紧缩模式开始时。这里有好消息和坏消息：在开发中你可以不去处理这些问题。如果这样，你会在一次又一次生产环境中遇到它们。在开发中处理这些问题并不需要多少开销，不管是时间还是精力，但如果忽略它们，你的长期成本会远远超过这些。

请记住，你的程序会在有多个网络接口的服务器上运行。请确保它绑定了它所监听的套接字地址，并确保特殊的路由请求都设置并记录在案了。管理的功能应暴露在管理和监视网络中，而不是生产网络。

请确保使用虚拟 IP 地址访问集群服务，比如由其他系统提供的数据库服务器或网络服务。使用 VIP 能提供失效备援，不管有没有计划，都不需要你重新配置系统了。

应用程序应该作为应用程序用户运行，而不该是 root 用户或管理员用户。敏感的配置参数，比如数据库密码或密钥，需要单独保存在配置文件中。

并非每个系统都需要 5 个 9 的可用性。随着可用性级别的提高，会需要更高的成本。在项目发起人的帮助下，通过权衡成本和收益来考虑可用性需求吧。

不要从系统整体来定义可用性，我倾向于按系统的具体功能来定义。一定要记得将外部系统带来的可用性损失排除在外。

负载均衡和集群是高可用性的两个先决条件。你能使用多种技术，当然不同技术成本不同。有了可用性需求，你能应用各种负载均衡和集群方案以有效的成本满足需求。每个解决方案都有独特的考虑，因此这样定义高可用架构能使开发和部署更容易。

你的应用程序管理员绝不会和你一样对系统内部如此了解。通过让应用程序易于配置，来帮助操作人员减少犯错。这意味着分离必不可少的基础工作，比如 Spring 中与环境相关的 `beans.xml` 文件。将它们混合等同于把弹射椅按钮放在收音机旋钮旁边，迟早要坏事。

花时间让你的程序操作更简单些吧。启动和关闭应该不烦扰用户，任何管理功能也应该能被脚本化。漂亮的 Java 桌面管理 GUI 能帮助新人上手，但是没有人愿意数千次地点击这些漂亮的 GUI。

Part 4

第四部分

运营

本部分内容

- 第 16 章　案例研究：惊人的宇宙
- 第 17 章　透明度
- 第 18 章　适应

第 16 章
案例研究：惊人的宇宙

16.1 旺季

在 16 世纪中期，一个名为 Aloysius Lilius 的卡拉布里亚医生发明了一个新日历来修正广泛使用的儒略历的错误。儒略历有累积的偏差，几百年下来，冬至日的时间比实际提前了好几个星期。Lilius 的日历使用了一个更正与反更正的复杂系统，来保持官方日历中春分和冬至时间接近于天文事件的时间。每个 400 年的周期，日历日期的变化达到 2.25 天，但它们是可预测的周期性的变化。总体而言，错误是循环的，但是不累积。这个日历，由罗马教皇格里高利十三世下旨，而最终被所有欧洲国家采用，后来，甚至埃及、中国、韩国和日本也采用了（后三个国家做了一些修改）。有的国家最早在 1582 年就使用了格里历，大部分国家在 1920 年代采用。

由天主教会创建历法并不奇怪。像大多数历法一样，格里历被用来纪念圣日（即节假日）。它还用来根据每年太阳的活动周期标记其他领域里有用的反复事件，比如农业等。但是实际上，没有任何业务要靠格里历来开展。商业界使用日期方便地标记内部商业周期。

每个行业都有自己的内部年历。对于保险公司，每年都围绕“公开注册”开展，所有的计划都根据公开注册期来制定。花店的考虑都围绕着母亲节和情人节。这些碰巧标记在公历的具体日期上，但是在花店的思维和延伸的整条供应链中，这些季节有它们独特的意义，而并不与官方日历相关。

对于零售商来说，一年的开始和结束被美其名曰为“节日”。这里我们能看到各种宗教日历和零售日历间的对应关系。圣诞节、光明节和宽扎节都紧密相关。零售商高达 50%的年收入都发生在 11 月 1 日和 12 月 31 日之间。

在美国，感恩节，也就是 11 月的第 4 个星期四，[1]是零售业旺季的开始。按照长久以来的传

① 一些零售商游说国会，要求政府将感恩节提前两星期。

统，因为距那个时间点只有不到 30 天，消费者就开始认真考虑购买礼物了。显然，限期的压力跨越了宗教界限。购物者的恐慌，导致了一个集体现象“黑色星期五”。零售商通过各种办法鼓励和加强这种现象，调整分类、增加库存、猛打广告。实体店的访问量可以一晚上翻两番，在线商店可以增加 1000%。这不愧是真正的负载测试。

16.2 婴儿的第一个圣诞

我的客户在夏天推出了一个新的网上商店。推出之后的数星期和数月里，事实一再地证明为什么推出一个新网站就像生一个婴儿。你必须想到某些事情，比如半夜被叫醒，经常有一些可怕的发现（比如，“天哪！你拿什么喂养这个孩子？橙色培乐多彩泥？”或者“什么？为什么会在页面渲染的时候解析内容？”）。尽管网站推出之后有了各式各样的经历，我们还是带着谨慎的乐观接近旺季。

我们的乐观来源于这样几个因素。首先，我们在生产环境几乎加倍了服务器数量。其次，我们有确凿的数据表明网站在当前负载情况下很稳定。一些突发事件（主要是定价错误的项目）让流量攀升，流量顶峰高得足以让页面开始延迟，因此我们清楚多大的负载会让网站瘫痪。我们乐观的第三个原因是相信我们能处理网站遇到的任何问题。由于应用服务器的内在能力和我们为服务器构建的工具，相比以往我从事的任何网站，我们对网上商店内部的控制更强，可见性更好。这应该是决定我们是得到一个成功的感恩节周末，还是一场不折不扣的灾难的关键。

我们中的一部分人在劳动节放假期间把周末的任务完成了，所以周末可以轻松一下了。我利用 4 天的休假带上我的家人去外地的父母家，一起共进感恩节晚餐。我们也在周末安排了 24 小时的现场值班。正如我所说的，我们谨慎地乐观。请记住，我们的技术团队就在本地，主要的网站运营中心（SOC）在另一个城市，一天 24 小时都有熟练的工程师盯着。通常，他们在夜晚和周末会一直监视和管理网站。本地团队为 SOC 提供备份，如果遇到没有已知解决方案的问题时可以提供升级路径。我们当地的团队规模太小，无法像现场那样全天 24 小时工作，但是我们摸索出方法在感恩节周末这段时间全力以赴。当然，曾经当过童子军的我，为了以防万一，将笔记本塞进了已经整装待发的车里。

16.3 切脉

星期三晚上一到达父母家，我立即在他们的家庭办公室里打开了笔记本。只要有宽带和手机，我可以在任何地方工作。父母家有 3MB 的宽带，我用 Putty 这个我最喜欢的 SSH 客户端，登录到主机，然后启动采样脚本。当初在网站的准备阶段到发布的过程中，我参与了负载测试。大多数负载测试在测试完成后都提交结果。由于数据来自于负载产生器而非测试的系统内部，因此它算是个“黑盒”测试。为了从负载测试获得更多信息，我使用应用服务器的 HTML 管理 GUI 来检查一些重要数据，比如延迟、可用堆内存、主动的请求处理线程和活动会话。

如果你事先不知道要查找的东西，那么 GUI 是探索系统的好接口。[①]如果你知道你想要什么，那么 GUI 就多余了。如果你需要一次查看三四十台服务器，GUI 就完全不切实际了。

为了从负载测试中获得更多信息，我写了一套 Perl 模块来管理 GUI，它将 HTML 解析成有价值的信息。这些模块会让我获取和设置属性值，并激活应用服务器组件中的方法，包括内置和定制的方法。因为整个管理 GUI 是基于 HTML 的，所以应用服务器并不知道 Perl 模块或网络浏览器的区别。有了这些 Perl 模块，我能够构造一套脚本来对服务器的重要统计数据进行采样，隔一段时间就打印出详细信息和总结结果，不断循环。

这些都是简单的指标，但是自从网站推出以来，我们都是通过这些数据得知网站节奏和脉搏正常。只需看一眼，就能知道在 7 月份星期二的中午什么是正常的。如果会话数量上升或下降了，如果订单数错了，我们就会知道。用这种方法去发现问题，确实速度惊人。各种检测技术[②]提供了一张巨大的安全网，遇到问题时能够立即发现，而人类大脑的模式匹配能力似乎是出类拔萃的。

16.4 感恩节

感恩节的早晨，我一起床，连咖啡都顾不上喝，就跳进工作间来检查运行了一晚上的统计窗口。我简直不敢相信自己的眼睛。凌晨的会话数量就已经和平常最繁忙日子的高峰时间一样了。订单数太大了，我甚至打电话给 DBA 去确认订单是否有重复提交的。结果并没有重复提交。

中午时候，客户的订单达到了平时一周的量。我们响应时间和网站性能的综合指标——页面延迟，承受了压力但是问题不大。更好的是，它随着时间推移保持稳定，即便会话和订单数量一直在增加。我愉快地享用着火鸡晚餐。到了晚上，我们一天接到的订单已经接相当于本月累积的数量了。到了午夜，我们接到的订单已经相当于整个 10 月份的总量。网站挺过来了，它通过了第一次最严酷的负载测试。

16.5 黑色星期五

第二天是黑色星期五，早餐后我去瞧了一下统计数据。订单数量比头一天更多，会话数量也上升了，但是页面延迟仍然大概是 250 毫秒，这和我估计的恰好吻合。我决定和我妈出门去买些咖喱鸡原料。（本来是用感恩节的剩菜做晚饭，但是我想周六做咖喱，而周六我最喜欢的泰国市场会关门。）

① 我们用的 ATG Dynamo 是个特别的应用服务器，拥有最好的管理 GUI。它并不像 WebLogic 或 WebSphere 那样漂亮，但是它暴露了应用服务器的各个组件。就像原厂的大众甲壳虫发动机一样，你可以看到每个部分，它们如何工作，以及如何连接到其他部件。ATG 多年来一直在从事依赖注入方面的工作，比 Martin Fowler 提出“依赖注入”这个名词还要早，更是远远早于 Spring 的出现。

② 参见第 17 章“透明度”中对各种检测技术的讨论。

当然，如果一切顺利，不出大事，我是不会讲这个故事的。而且，不等我远离了我的电脑，事情也不会变得那么可怕。果然，我在半路上接到了电话。

“早上好，迈克尔。我是网站运营中心的丹尼尔。”对方说。

“不会是出了什么问题吧？”我问。

“SiteScope 当前所有的 DRP 都呈红色。我们一直回滚重新启动 DRP，但是它们随即就失败了。大卫正在开个电话会议，想要您加入进来。”

在这样简单生硬的对话中，丹尼尔告诉我网站停机了，而且问题很严重。我们的外部监测工具 SiteScope 以真实用户的方式访问网站（见图 16-1）。只要 SiteScope 变红了，我们就知道客户不能买东西了，我们正失去收入。在 ATG 网站，[1]页面请求由服务器实例处理，这些实例只是服务于页面请求。网站服务器通过 DRP（Dynamo 请求协议）来调用应用服务器，因此很自然用 DRP 来代表请求处理实例了。红色的 DRP 表示请求处理实例对页面请求停止响应了。“所有的 DRP 都是红色”表明网站停机了，网站正以每小时 100 万美元的速度损失订单。“滚动重启”意思是他们正尽快地关闭和重启应用服务器。在一台主机上启动所有的应用服务器实例需要 10 分钟。你一次最多能启动四五台主机，但是花的时间就不止 10 分钟，因为数据库服务器的响应时间开始受到影响，这让启动过程花的时间更长。结果，如同木船破了个大洞，大家拼命往外舀水也无济于事。

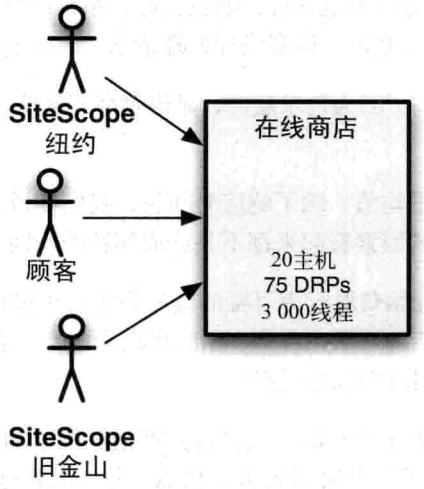


图 16-1 SiteScope 模拟用户访问前端存储

“好，我现在就打进来，但我要 30 分钟才能回到电脑旁。”我告诉他。

① 一个 J2EE 应用服务器，非常合适电子商务应用。参见网站 http://www.atg.com。

丹尼尔说："我给你电话会议拨号密码。"

"没关系。我记得起密码。"我说。

我打进电话，听到一片嘈杂声。显然，连入网桥的有会议室里的一个免提电话。从这回音不绝的会议室里要分辨出 15 个人的声音真是困难，况且还有人不断地连入退出，说些他们以为有帮助的信息——"网站有问题了。"是的，这个我们知道。谢谢，请挂掉吧。

16.6 重要的信号

事情发生在丹尼尔打电话给我之前的 20 分钟。运营中心已经升为现场了，运营中心经理大卫决定让我加入进来。

为了客户，也顾不得打扰我的假期了。此外，我先前告诉过他们，需要时就打电话给我。

此时，我得知了事故发生的这 20 分钟里的一些事情。

- 会话数非常高，比头一天的还要高。
- 网络带宽使用率很高，但并没有达到极限。
- 应用服务器页面延迟（响应时间）很高。
- 网络、应用程序和数据库 CPU 使用率很低，相当低。
- 搜索服务器，这个常见的罪魁祸首，响应良好。系统统计数据看起来没问题。
- 请求处理线程几乎总是繁忙。许多线程处理请求的时间超过了 5 秒钟。

事实上，页面延迟并不高。因为请求超时，显得效率非常低。统计数字只显示了完成的请求的平均数。

未完成的请求没有计入平均值。除了响应时间长，我们知道大概是因为 SiteScope 没能完成模拟事务，其他我们常怀疑的因素看起来都不是造成问题的原因。

为了获得更多信息，我开始获取行为可疑的应用服务器线程转储。我让阿什霍克检查一下后端的订单管理系统，他是我们的明星工程师，当时也在会议室。他发现后端也有和前端类似的状况：低 CPU 使用率，多数线程都长时间繁忙。

从我接到电话，将近过了 1 个小时，或者说是网站停机了 90 分钟。这意味着我们既损失了订单，也离未达到 SLA 很近了。我恨不能达到 SLA。我个人是这样，所有的同事也一样。

16.7 诊断测试

前端应用服务器的线程转储显示了类似的 DRP 状况。一些线程忙着调用后端，而其他的线

程大多等待着后端的可用连接。等待线程都阻塞在资源池中，没有超时。如果后端停止响应，那么调用的线程不会返回，这些阻塞的线程再没有机会进行调用了。总之，每个请求处理线程——总共有 3000 个——什么事都没干，这就能解释为什么 CPU 占用率低了：所有的 100 个 DRP 都是空闲的，都在等待一个不会到来的响应。

将注意力转到订单管理系统。系统的线程转储反应了 450 个线程都被用来调用外部集成点，如图 16-2。你很可能猜测到，所有的其他线程都被阻塞，等待着调用外部集成点。该系统处理的是送货上门的物流调度。我们立即呼叫这个系统的运营团队。（它由一个不同的组管理，他们没有每天 24 小时都在的技术支持人员，而是轮流应召值班。）

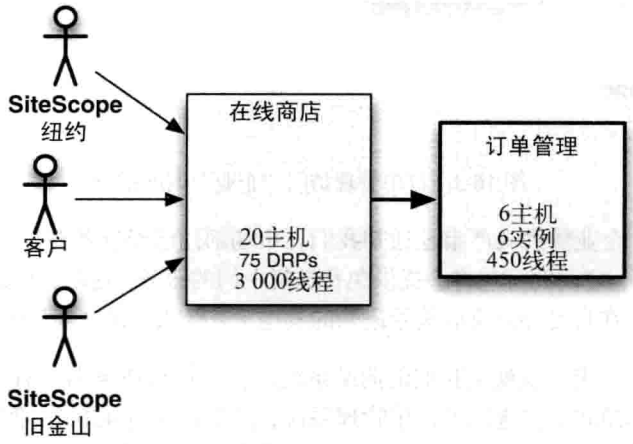


图 16-2 前端存储访问后端订单管理

我想这是妻子给我带来一盘火鸡剩菜和食物的时候了。在等状态报告的空闲里，我将电话静音，然后快速地吃完饭。这时，我手机的电用光了，而且无绳电话也快没电了。（我不能使用普通电话，因为耳机插头用不上。）我祈祷着，在无绳电话没电之前，我的手机能充到足够的电。

16.8 专家打来电话

当那个技术支持工程师打电话进来的时候，虽然仅仅过了半个小时，但对我来说，就像过了半个世纪！他解释说 4 台通常处理调度的服务器，有两台在周末假期停机维护，还有一台出现故障且原因不明。直到今天，我都不知道为什么他们不早不晚偏偏安排在那个周末做维护！图 16-3 展示了这 3 个系统的大小。

那台唯一剩下的服务器在变慢然后挂掉之前最高可以处理 25 个并发请求。我们估计，订单管理系统很可能在那时发送了 90 个请求。果然，通话的工程师检查这台调度服务器时，它的 CPU 使用率达到 100%。别人好几次呼叫他请求高 CPU，但是都没有得到回应，因为以前这个组经常

被呼叫要 CPU 瞬时高峰使用率时，都证实为了误报。“狼来了”喊多了他们也就麻木了。

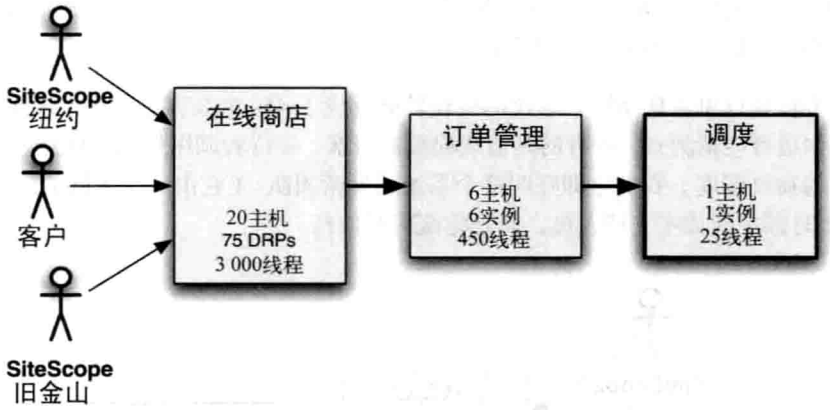


图 16-3 订单管理访问“企业”调度系统

在电话会议上，企业赞助人严肃地通知我们，市场部门已经准备在周五早上的报纸上插入新的广告，告诉用户周一之前的订单将会提供免费送货上门的服务。这时，整条线上，包括会议室话筒前面的 15 个人，和在自己办公桌前拨号进来的其他十几个人，在这 4 个小时里头一次一片沉默。

总之，这个网上商店，也就是我们的前端系统，在 100 台服务器上有 3000 个线程，而且面临着突然巨变的流量模式。它淹没了订单管理系统，这个系统有 450 个线程，既要处理前端请求也要处理订单。订单管理系统淹没了调度系统，后者只能同时处理 25 个请求。

这将会持续到周一。真是一场噩梦。网站都已经停掉了，我们却没有应对的预案。在这紧要关头，我们不得不想出一个临时的解决方案。

16.9 比较解救方案

开始头脑风暴。很多的建议都被否定了，主要是因为应用程序的代码行为在这种情况下是未知的。很快大家就清楚了，唯一的答案就是停止这么多的请求，来检查调度的可用性。由于周末发起了免费送货上门的营销活动，我们知道用户的请求不会减少。我们必须找出应对的方法。订单管理系统没法做到这点。

我们看了看商店的代码，有了一丝希望。它使用了标准资源池的子类来管理订单管理的连接。实际上，对于调度请求它有自己的连接池。我不知道为什么要设计单独的连接池，很可能是根据 Conway 法则，但是它挽救了这个周末。因为它有这些连接的组件，我们能将这个组件作为我们的截流阀。

如果开发者增加了一个 enabled 属性，那么可以很容易地将它置为 false。虽然下次我们能找到更好的做法，但是这次只能这样了。将最大数量设为零能禁用资源池。我问开发人员，如果资源池返回 null 而不是一个连接会如何？他们回答代码会处理这种情况，会提供给用户一条礼貌的信息说明送货调度暂时不可用。很好。

16.10 条件是否会响应处理

我的一个 Perl 脚本能在任何组件中设置属性值。作为实验，我使用这个脚本来将资源池（仅仅是 DRP 的）的最大值设为 0，然后我设置 checkoutBlockTime 为 0。什么都没发生。行为上没有任何改变。然后我记得只有在池启动之后，最大值才有效果。

我又使用了一个脚本，用来在组件中调用方法，我用它来调用 stopService() 和 startService()方法。瞧！DRP 又开始处理请求了！太高兴了。

当然，因为只有一个 DRP 开始响应了，负载管理器开始将每个页面请求都发送给这个 DRP。结果它就像世界杯比赛场边唯一开门的啤酒摊，立马被挤爆了。但是，至少我找到了一个方法。

我运行我的脚本，这次标记为“所有 DRP”。它们都设置 max 和 checkoutBlockTime 为 0，然后重启服务。

重启组件，而非整个服务器，是面向恢复的计算的核心概念。虽然我们没有 ROC 建议的自动化级别，但还是能在不重启服务器的情况下恢复服务。如果需要改变配置文件，然后重启所有服务器，在这种负载级别下至少要花 6 个小时。动态的重新配置和重启连接池只花了 5 分钟都不到（只要我们知道该怎么做）。

执行完脚本之后，我就立刻看到用户流量进来了。页面延迟也开始下降。大约 90 秒钟以后，SiteScope 中的 DRP 呈现绿色。网站恢复运行了。

面向恢复的计算

面向恢复的计算（ROC）项目是伯克利和斯坦福大学的一个联合研究项目。[1] 该项目的基本原则如下。

- 故障不可避免，不管是硬件还是软件。
- 建模和分析不可能充分完成。所有的的故障模式都是不可先验的。
- 人的行为是系统故障的主要来源。

他们的研究有悖于系统可靠性的大多数工作。虽然大多数工作关注的是消除故障来源，但是 ROC 认为故障不可避免地会发生，这也是这本书的重要主题！他们的研究也主要是为了提高系统面临故障时的生存能力。

ROC 的很多概念都在今天的各种语言和平台上实现。跟随他们关注危害范围、自动故障检测、组件级的重启，你会获得巨大的收益。

① 参见http://roc.cs.berkeley.edu/。

16.11　收尾

我写了一个新的脚本，它能完成重置连接池的最大值所需的全部操作。它设置了 max 属性，停止服务，然后重启服务。只需一个命令，运营中心或在“指挥所”（也就是会议室）的工程师就能把最大连接数重置为想要的任何值。我后来得知，这个脚本在周末不断使用。由于将最大值设为 0 禁用了送货上门，企业赞助商希望当负载较小时将值设大，而负载很重时设为 1（而不是 0）。

我挂上电话，然后将孩子们抱到床上。这用了好一会儿。他们告诉我各种好玩的事情：去公园、玩洒水，还有看见后院的兔宝宝。我喜欢听这些。

第 17 章
透 明 度

船上有经验的工程师能够根据巨大柴油机的声音判断出哪里会有问题。和发动机相伴多年，他们已经学会了如何识别正常情况、微小故障和不正常状况。要知道，他们身处的环境中环绕着各种声音。当发生问题时，工程师掌握的引擎内连杆机构的知识可以指引他们快速准确地找到问题——仅仅依靠一两个线索——看起来好像有特异功能一样。

我们的系统并不能像那样自然而然地公开展示，它们在不可见的盒子中运行，没有可见的运转部件，风扇稳定的呼呼声很少传达出什么有用的信息（不过，如果风扇停止，我们知道一定有问题！）。硬件不在我们的房间里，经常放在另一栋大楼里，有时甚至在另一座城市。如果我们想要得到船上工程师实际获得的那种“环境意识”，我们必须把“透明性”建造到我们的系统中来。

透明性是指这样一种特性，它能让操作员、开发者和商业赞助商理解系统的历史趋势、当前状态、瞬间状态和未来设想。透明性系统可以传达信息，并且在传达信息时，它们训练人类操作者。巡航船只上的巨大柴油发动机向外传递信息，是通过环境噪声和震动，或通过定量的计量信息，或者在极端情况下（一般是糟糕的）通过冒烟来做到的。

在调试“黑色星期五问题”时（参见 16.1 节“旺季”），我们依靠组件级的可见性来查看系统当前的行为。这种可见性不是偶然因素，它是基于透明性实现并反馈于心的产品。没有这种级别的可见性，我们可能能够指出网站很慢（如果有个不满的用户呼叫我们，或者企业中有人碰巧点击了网站），但是却不知道为什么。就像你的金鱼生了病——你帮不上任何忙，只能等待，生死由命。

透明性具有实践上和心理上的好处。在实践上，调试透明性系统非常简单，所以相比不透明系统，透明性系统能更快成熟。从心理上来说，设想一下，每年坐下来一次，让你的将来取决于掷骰子的结果，骰子各个面标记为“奖金”、“升职”、“解雇”、“大量加班”、“被人轻视”和“同事表扬”，但是由别人来投掷或者特意将一个面向上放置。你不知道，也不能控制。任何对系统成功负有责任的人——从应用程序开发者到业务分析师到项目赞助商——他们的未来很大程度都依赖于系统的行为，但是却都不知道会是什么结果。

如果必须增加容量，你完全依赖于从现有基础架构收集的数据。你需要结合技术数据和商业度量信息来理解系统的历史和当前状态，才好预测未来。良好的数据有助于做出良好的决策。没有可信赖的数据，你的决策就会基于别人的政治影响力、偏见甚至发型。

最后，没有透明性的系统不能在生产中长期生存。如果管理员不知道它正在做什么，就不能调整和优化；如果开发者不知道在生产环境中什么能行和什么不灵，就不能增强它的可靠性或者适应性；如果赞助商不知道在它上面是否赚钱，就不会赞助未来的工作。没有透明性，随着每次发布，系统将逐渐衰败，运转得更差。系统能够成熟，当且仅当它们有某种程度的透明性。

本章讨论四种层次的透明性：历史趋势、未来预测、当前状态和瞬间行为。还检验了能够实现透明性的现有技术，和有待开发的技术。一些厂商发现了 Web 系统性能测量和报告上的商机，因此当前掀起了一股创建综合“企业级应用管理”软件的风潮。这是一个富矿，所以我会比平常更多地讨论具体的产品。很可能当你读这本书时，这些公司将已经被并购到一些大企业中去了。

17.1 视角

不同的选择要求不同的视角。所有这些层次并不能使用相同的系统观点。如同“天气怎么样？”这个问题，对于一个园丁、飞行员和气象学家分别意味着非常不同的事情。如果 CEO 和系统管理员都问“事情进行得怎么样？”，要知道他们指的是完全不同的事情。

历史趋势

历史是一个相当可塑的词。用“上一周”去定义“历史”的场合可是真不多，这可能是 IT 运营和时尚之间唯一的联系了。

尽管这显得非常短视，但我们不得不关注历史观点。虽然有人秉持华尔街风格的否定论，[①]但我们还是有可能通过检查历史记录来预测未来的系统行为。不管是业务度量信息（顾客、订单、转换率、收入、运输成本等）还是系统度量信息（可用存储、平均 CPU 利用率、网络带宽和错误日志记录数），这么做都是可行的。

显而易见，一段时间内的历史记录必须存储在某处。数据库最能提供历史视角。17.2 节“透明度设计”以 OpsDB 提到了这点。OpsDB 可以被用来调查异常或者趋势，因为它包含了系统和业务级的度量信息数据，所以可以被用来识别实时及跨层次的相互关系。

历史视角可以通过表格、图表、微软 PowerPoint，或者分析报告展现。它缺乏当前状态的实时性，因而不适合通过仪表盘显示。

① 你知道的，他们说“过去表现的不能保证将来的结果”。

因为可以用来发现新的和有意思的关系，所以历史数据应该能够通过多种工具访问，比如微软的 Access 和 Excel。如果你的公司已经购买了一个商业智能（BI）或者报表工具，那么它无疑应该使用历史数据。只是需要注意不要让这些工具直接去访问生产事务数据库。（参见 9.8 节中的“历史数据”。）

以下这几个常见问题都具有历史视角。

- 我们昨天有多少订单？
- 和去年的这一天相比，结果如何？[①]
- 第一个季度我们使用了多少磁盘空间？
- 上一次我们遇到流量高峰时，哪个系统是限制因素？
- 过去三年中，顾客流量的增加和 CPU 使用率增加相比，结果如何？

预测未来

在 *Planning Extreme Programming*[BF01]一书中，Kent Beck 和 Martin Fowler 让“昨天的天气”这个说法广为大家接受。复杂的天气预报系统被一个简单的算法击败——预测今天的天气和昨天一样的正确率将是 70%。他们讲了一个有趣的故事。虽然说天气预报的理论基础非常复杂，但是这个说法很好地体现了这样一个认知：我们可以通过检查历史来预测未来。

在我们自己的系统中，面向后面的历史信息同样也是我们的水晶球。未来的预测几乎总是关于联系和关联的，而不是直接的测量。至此，最常见的问题就是：“我们的容量是什么？”（见 8.1 节“定义容量”，以及 8.3 节“关联”）。尽管这看起来是一个关于单一度量的问题，但是其中暗含的判断条件是：“后面的 N 个用户是否对前面的 N 个用户有一定比例的影响？”

预测总是建立在系统模型之上。糟糕的预测来自糟糕的系统模型（比如，线性关系）。真正优秀的模型很难开发，它要求人们对某些课题有深厚的理解，比如队列理论和随机模型，需要运筹学专家。这些人能够建造一个模型，只需要架构图就能非常精确地预测系统的性能。我不确定对于电子商务系统这类已经规划良好的领域再花费精力（和费用）是否值得，但对于全局通信网络建模这类工作还是很合适的。

另外，通过在过去的数据中查找关联，我们有可能建立“足够好的”模型——在能够应用的相关领域中[②]去做预测。这些相关模型可以建成电子表格，让技术背景不强的用户能够直接进行各种情况的预测。

预测未来使用的一般都是敏感信息，它们也没有即时数据的紧急性，因此，一般不应该构建到仪表盘中。

① 注意，“这一天”不总是意味着日历上的同一天——它几乎总是对应着一个星期中的同一天而不是一个月中的同一天。

② 请注意牛顿运动定律也只是适用于特定领域……但是却能够让飞船经过 20 年的飞行到达海王星。

应用的新发布能改变或破坏预测数据使用的那些关联。因为这些预测数据像系统行为的衍生，它们必须在每次应用发布后重新检验，以确定是否依然可行。（当然，在做出判断之前你必须建立一个适当的新测量主体）这也意味着基于这些数据预测的任何事情都应该指明使用了哪些预测数据集。[①]

这里是具有未来视角的一些常见问题。

- 我们每天能够处理多少顾客？
- 什么时候我得买更多的服务器（或者磁盘、带宽，或者其他任何计算资源）？
- 这个假日季节我们能行吗？（注意这需要对预测再做预测，误差概率不是倍增而是平方。）

当前状态

要理解当前状态和此刻的行为之间的区别，可以设想一个正在慢跑的胖子。此刻行为是为了提高健康水平，但是他的当前状态可能是立刻会犯心脏病。

“当前状态”描述了系统的整体状态。它并不是关于正在做什么以及已经做了什么，而应该描述每一块硬件和每一个应用服务器、应用程序和批处理作业的状态。

每一个组件的状态用事件和参数的组合来定义。事件是在各个时刻发生的很多事情，有些是正常或必需的事情，而有些则揭示了不正常的问题。比如每天的库存反馈就是一件必需的事情。我们预期这个事件应该发生，如果没有发生则要报警。不正常事件通常按“低–中–高”或者“警告–严重–紧急”来分类。

参数是可以从系统中观测到的连续度量信息或者离散状态。这正是透明性至关重要的地方。披露内部状态较多的程序提供的参数更精确和真实。最基本地，我们有操作系统的度量信息：CPU 利用率、可用内存、内存交换率、每个接口的网络带宽和可用磁盘空间。在应用服务器内部，一般都有大量可用的度量信息和状态。

- 内存

堆最小值、堆最大值、代的大小。

- 垃圾回收

类型、频率、占用的内存、请求尺度。

- 每一个线程池的工作线程

线程数、繁忙线程、繁忙时间超过 5 秒的线程、高水位（使用的最大并发线程数）、低水位、

① 我知道，这像个学者的语气。我不指望商业发起人会精确地看待他们的方法，但是我希望开发者和系统工程师去这样做！

线程不可用次数、积压的请求。

- 每个数据库连接池

连接数、正在使用的连接数、高水位、低水位、连接不可用次数、积压的请求。

- 每一个请求渠道的流量统计

处理的总请求数、平均响应时间、异常退出的请求数、每秒请求数、上次请求时间、当前是否接受流量。

应用程序自己应该披露足够多关于自身的度量信息。

- 每种类型的业务事务

处理数、异常退出数、美元数值、老去的事务、转换率、完成率。

- 用户

图表或者分类、技术图表、注册用户百分比、用户数、使用模式、发生的错误。

- 集成点

当前状态、手动覆盖、使用的时间、远程终端系统的平均响应时间、错误数。

- 环路断路器

当前状态、手动覆盖、失败调用数、上次成功调用时间、状态转移数。

就像你所看到的，即使一个中型程序，也可能有上百个参数。每个参数都有一个正常的可接受的值范围，可能是围绕离散值的一个容忍度，也可能是某个不应该超出的阈值。只要在可接受的范围内，参数就是正常的。通常还有另一个更紧的范围，表示“注意”信号，警示参数正接近它的阈值。

对于连续度量参数，正常情况的经验定义将是“这段时间有意义的值加上或减去两个标准偏差”。要选择参数开始变得有意思的那个时间段。对于大多数流量驱动的度量，具有最稳定关系的时间段会是“每周这个时刻”（hour of the week），比如星期三的下午 2 点。每个月几号则没有多少意义。在某些行业，比如旅游、装饰和体育，最相关的测量是从一个节日或事件向前数。

对于零售商，则是星期几再加上更强的每年第几周。各个公司都有自己的特点，不能一概而论。[①]

① 查查“自动工作量特征”（automatic workload characterization）的有关文献就能获得各种生动的描述。虽然表达起来可能干巴巴的，但一场激烈的争论已经展开多年了。

声名狼藉的仪表盘

状态通常以熟悉的“红–黄–绿”图表展现在仪表盘上。将一个复杂的、多层次的分布式系统的全部状态归类成三种颜色的系统，似乎过于简单化了，但它确实有易于别人熟悉的优点。如果每个人都对颜色的含义有相同的认识，这么做就会有帮助。图 17-1 显示了过去我遵从的理解。这个定义还表达了一个容易被人忽视的问题：好事过头反成坏事。

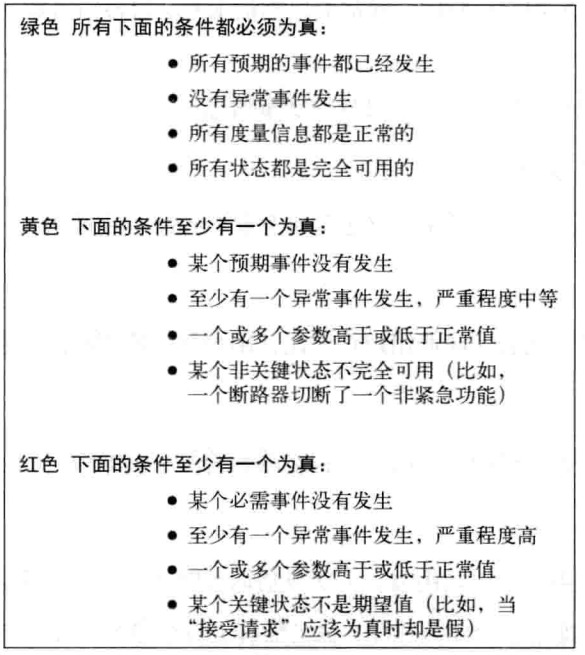


图 17-1 颜色的含义

系统的当前状态显然仪表盘显示起来是容易的（它实际上定义了一个仪表盘）。仪表盘应该是广泛可见的，将它投影到餐厅的墙壁上没有任何问题。越多的人理解系统正常的行为越好。从最实用的角度说，仪表盘应该能够对不同的用户展现整体系统不同的方面。运营工程师可能首先关心组件级别的视角，开发者可能希望应用级别的视角，然而商业赞助商可能希望围绕功能或者业务流程级别的视角。显然，这意味着仪表盘应该知道这些不同视角的联系。当观测到组件级别的输出时——比如，网络失败——管理员应该能够看到哪些业务流程会受到影响。这有助于和赞助商交流，并为问题确定一个合适的优先级。

大多数系统都有规律性的预期事件。它们可能来自其他系统，或者发布到其他系统，或者仅仅是集成到遗留系统的批处理工作。无论什么用途，这些工作都是系统的一部分，就像 Web 服务器或者数据库服务器一样。它们的执行属于“必需的预期事件”。仪表盘应该能够反映这些预期事件，它们是否发生，它们是否成功。

当前状态对于和系统的日常功能有经济关系或职业关系的任何人都是很重要的。IT 人士显然属于这个范围，当然也有 IT 圈外的赞助者，其职业也与系统紧密关联。通过在不同的时间和不同的条件下观察系统的当前状态，所有这些人都将具有自己的职业意识。

即时行为

即时行为回答了这样的问题："xxx 现在怎么样？"当意外已经发生时，人们对即时行为最感兴趣。

显然，即时行为和当前状态相关。即时行为中的异常往往来自错误状态。日志文件中的栈、线程转储、抛出的错误，和对用户的不可用响应，都是行为问题的例子，它们有可能会也可能不会在状态视角中显示出来。例如，用户不喜欢抛出错误。这种情况发生时他们趋向于离开。结果，向用户输出错误将最终导致"每小时事务"度量信息掉落到正常范围之下。最好在用户走开之前捕获这些异常行为。

即时行为属于监控系统的范畴。从自用的日志文件扩展到数百万美元的 HP OpenView，监控系统坐在你系统的外部，就像老大哥一样监管你与计划保持一致。

这也是线程转储的范畴（见 2.3 节）。像 JMX 这样的框架也能够观察即时行为，因为它们允许管理员查看运行程序的内部状态。

哪些人来查看即时状态呢？由于它们的潜在危害性，某些部分必须受到限制。不是每一个人都能访问 JMX 终端，因为他们可能在做有害的事情，比如关闭服务器或者改变关键参数。有些收集状态的方法（比如，线程转储）需要访问服务器的权限。除了这些明显的有害情况，还要警惕当开发者或者商业赞助者要求查看系统的即时行为时，运营部门经常感到受到威胁，特别是处在缺乏信任甚至是不友好的氛围中的时候。运营部门担心他们会被每一个信号和异常导致的大量询问淹没。更糟糕的是，他们害怕他们的客户——系统的付费者——先于他们发现问题。设想某个市场总监打电话过来问"为什么其中某个服务器的 CPU 使用率比其他的更高？"，你还只能回复"我不清楚，这一整天我还没有看图表呢"，再没有比这样的事情更令人感到尴尬了。

缓和这种紧张情绪的一个方法便是提高现有状态的透明度。当赞助者询问更多信息时，他们通常想要得到的都是状态信息而不是即时行为。（这就是我为什么最初将这两种问题分离开来的原因。）站在他们的角度来看，这完全是可以理解的。他们需要对该系统的财政状况负责，因此必须了解系统的现状。但是，他们其实并不需要知道某个应用程序每一次在日志文件里遗留了什么信息。他们需要知道的是"我们的收入是否符合预期？"或"上一次行动是否增加了转化率？"，或者"我们应该购买更多的检票机或安置更多的代理商吗？"。这些疑问通常是通过一个状态仪表盘、历史趋势图和未来预期来回答的，而不是通过注视 SNMP 陷阱和线程转储来得到答案。

17.2 透明度设计

透明度来自优雅的设计和架构。在开发后期“增加透明度”就如同“增加质量”一样效果很小。可能可以完成，但是比开始开发要花费更大的精力和代价。

在应用和服务的内部增加可见性还不够。严格的局部可见性导致严格的局部优化。比如，零售商运行一个大项目来让网站快速显示条目。当需要在接近午夜时分完成的时候，更新却一直做到凌晨五六点钟。项目优化了向网站提供内容的批处理任务，并达到了目标，让批处理任务提前两个小时完成。然而，必须等到凌晨五六点钟一个长时间运行的并行处理结束之后，条目才能出现在网站上。批处理的局部优化对于全局来说没有什么用处。

一次只有一个应用的可见性也会掩盖扩展效应带来的问题。例如，观察到一个应用服务器的缓存刷新，并不能透露出每个服务器正把其他所有服务的项目踢出缓存。每次显示一个项目时，它是偶然被更新，因此导致向所有其他服务器发送一个缓存失效通知。一旦所有缓存的统计显示在一个页面上，问题就显而易见了。没有可见性，我们只能添加很多服务器来达到必要的性能，但是每个服务器又会使问题变得更加严重。

在透明性设计中，要密切注意耦合。监控框架能相对轻易地侵入到系统的内部。可使用下面讨论的标准避免这种过度的紧耦合。监控和汇报系统应该像一个建立在系统周围的外骨骼，而不是缠绕在系统内部。特别是，关于什么度量信息应该触发警报，在什么地方设置阈值，以及如何将状态变量汇总成系统的总体健康状态，这些决定应该放到应用程序的外部去做。这些是政策性决定，和应用程序代码的变动要分开。

17.3 使用各种技术

运行在服务器上的进程本来是完全不透明的，除非你对它运行调试器，否则它不会特别地披露任何东西。它可能工作正常，可能正在运行最后一个线程上，也可能在循环往复地做无用功。就像薛定谔的猫，除非你注视它，否则不可能知晓进程究竟是活的还是已经死了。

所以首要的就是从外面探究线程里的信息。本节来考察一下能减少进程边界不透明度的重要技术，它们可以归类为“白盒”技术和“黑盒”技术。

黑盒技术运行在进程外部，通过外部观察到的现象检查进程。黑盒技术可以在系统发布后实施，通常由运营部门执行。就算要观察的系统对黑盒技术一无所知，在开发阶段你依然可以做些事情，来方便这些工具的使用。

相反，白盒技术运行在被观察对象的内部——不论是一个进程还是整个系统。系统可以通过这些工具刻意地披露一些信息。这些必须在开发阶段就集成进来。与黑盒技术相比，白盒技术有必要与系统更紧密地耦合。

17.4 日志

尽管大家花费了几百万美元的研发费用，去开发“企业级应用管理”软件包，建设拥有可显示网络结构彩图的大号等离子显示器的漂亮运营中心，但是，好的古老的日志文件依然是最可信赖的、最通用的信息媒介。身处 21 世纪，蓦然回首，发现日志文件依然是最有价值的工具之一，不由令人莞尔。

日志无疑是一种白盒技术，它处处都集成在源代码中。日志因为以下这些好理由而广为使用。

日志文件反映了程序内部的活动。因此，它们披露了那个程序的瞬间行为。它们还是持久化的，因此可以诊断它们来理解系统的状态，尽管这通常需要一些“领悟力”来追踪为何转换到当前状态。

如果你想避免和个别监控工具或框架的紧耦合（见 17.5 节中的“商业监控系统”），那么日志文件就是你要用的方法。没有什么比日志文件耦合更松了，每个现有的框架或者工具都能处理日志文件。松耦合意味着日志文件在开发中也是有价值的，这时候你不可能有 OpenView 或类似工具。

虽说价值很大，但是日志文件还是被大量滥用了。这里是一些成功使用日志的关键点。

配置

尽管开发者的直觉是将日志目录（`logs`）放到应用程序的安装目录中，但管理员通常希望把日志放在不同的“轴”上（硬盘或者 LUN），与操作系统和内容分开。日志文件可能很大。它们增长很快，并占用大量的 I/O。把它们放到单独的驱动器上能并行使用更多的 I/O 带宽，并消除对繁忙驱动器的竞争。

如果你把日志文件路径做成可配置的，那么管理员就可以设置文件路径。如果路径不可配置，那么他们只能迁移文件，但是你可能不会喜欢这么做的方式。

在 UNIX 系统上，symlink 是最通用的变通方案。这需要创建一个从 `logs` 目录到实际文件路径的软链接。每次打开文件时有一点 I/O 开销，但是相对于繁忙磁盘上的竞争这点代价还是很低的。我还见过用于日志的文件系统，直接挂接在安装目录下。

幸运的是，所有的通用日志框架都支持配置路径。（这又是不要自己迁移目录的一个小小理由。）

日志级别

当人类阅读（哪怕是仅仅扫视）新系统的日志文件时，他们在学习“normal”对系统意味着

什么。一些应用程序，特别是年轻的电子商务应用，充满了噪音，它们在日志中产生了很多错误。有些则是安静的，在正常操作中不报告任何东西。无论哪种情况，人类都慢慢适应了这些，知道这就是健康或者正常的。

很多开发者实现日志的时候，就像他们自己是日志文件的首要顾客。实际上，运营部门的管理员和工程师在这些日志上要花费更多的时间。日志应该针对产品运营，而不是以开发或者测试为目标。显然，任何级别为“ERROR”或者“SEVERE”的日志都需要运营部门采取行动。不是所有的异常都应该记录成一个错误。只是因为用户输错了信用卡号码，然后验证模块抛出了一个异常，并不意味着需要为此做点什么。将业务逻辑错误或者用户输入错误在日志中记录为警告（如果需要的话）。用“ERROR”标记严重的系统问题。例如，一个断路器错误的“open”是一个错误。这在正常环境下是不应该发生的，这可能意味着连接的另一端需要采取行动。连接数据库失败是一个错误——不是网络就是数据库服务器有问题。`NullPointerException` 不能自动成为一个错误。

在生产环境中去调试日志

在论及日志级别时，我要谈论一件令我气恼的事：在生产环境中去“调试”日志。这不是个好主意，它会产生大量的噪音，让真正的问题深埋在了许多方法踪迹或者无关紧要的检查点中。人们很容易在生产环境中打开调试消息，所需要的就是在日志配置开启了某些调试级别时提交一下 cvs 或 svn。我建议为你的编译过程添加一个步骤，自动删除任何开启了调试或者追踪级别的配置。

消息目录

运营部门经常要求提供系统产生的所有日志消息。过去这让我备受折磨。可是目前，有了像 Eclipse、IDEA 和 NetBeans 这样的集成开发环境，它们的国际化工具只需要几个小时就能整理出所有的日志信息。把它们收集到一个资源包，就很容易发给运营人员。

而且，你甚至可以为每条消息添加明确的代码。消息代码有两大优点。第一，它们可以让运营部门和开发部门之间准确地交流。如果有人跟你说“我不确定整个消息是什么，但是它提到了一个致命的系统错误”，那你就需要消息代码。第二，运营部门很容易在操作说明书或者知识库里面查找消息代码。假设你的代码中出现很多如下的语句：

```
try {
  ...
} catch (TimeoutException e) {
  LOGGER.log(Level.SEVERE, "Timeout failure connecting to fulfillment system." , e);
}
```

应用 Eclipse 的“Externalize Strings”产生一个新文件、messages.properties，以及一个新类 Messages。这里是 Eclipse 向 messages.properties 写入的内容：

code/transparency/messages.properties

```
FulfillmentClient.0=Connection refused.
FulfillmentClient.1=Authorization failed. Credentials refused.
FulfillmentClient.2=Timeout failure connecting to fulfillment system.
```

这些消息通过新工具类 Messages 的 getString(String key)来访问，由 Eclipse 在原始类的同一个包中创建：

code/transparency/Messages.java

```
public static String getString(String key) {
    // TODO Auto-generated method stub
    try {
        return RESOURCE_BUNDLE.getString(key);
    } catch (MissingResourceException e) {
        return '!' + key + '!' ;
    }
}
```

虽然“Externalize Strings”是为了让国际化更简单而创建的，但是你可以将它用于不同类型的接口。一旦 Eclipse 将所有的字符串收集到一个文件中，并带有一个共用方法返回它们，你就很方便去做些修改。例如，方法 getString(String key)的参数 key 只有两处小的调整，就能生成消息代码：

code/transparency/Messages.java

```
public static String getString(String key) {
    // TODO Auto-generated method stub
    try {
        return "(" + key + ") " + RESOURCE_BUNDLE.getString(key);
    } catch (MissingResourceException e) {
        return '!' + key + '!' ;
    }
}
```

人为因素

首先，日志文件是可读的。这意味着它们构成了一个人机界面，并且应该考虑人为因素。这可能听起来无关紧要，甚至可笑，但是在一个充满压力的形势下，比如 1 级严重的故障，人类错误地理解状态信息将拖延或者加重问题。在三里岛反应堆上的操作员误读了冷却液压力和温度数值，导致他们在处处采取了完全错误的行动（见 *Inviting Disaster*，[Chi01]，49 ~ 63 页）。尽管我们所做的大部分系统在遇到障碍时都不会泄露放射性水汽，但是它们也会导致上千美元的损失。因此，我们有义务确保日志文件向阅读它们的人传达出清晰、准确并能付诸实施的信息。

如果日志文件是一个人类接口，那么它们的写法要使人类可以尽快认出并解读。格式应该尽可能是可读的。人类视觉系统是一个高速和复杂的模式匹配机器，为什么如此多的厂商这么努力要去打败这种能力？图 17-2 显示了一个日志格式的例子，只能让计算机或者可能火星人读，人类无法读。这来自 WebLogic 9.2 的启动日志。你能以多块的速度发现其中的警告信息？

```
<Aug 13, 2006 7:24:53 PM CDT> <Notice> <Log Management> <BEA-170027> <The server
<Aug 13, 2006 7:24:54 PM CDT> <Notice> <WebLogicServer> <BEA-000365> <Server stat
<Aug 13, 2006 7:24:54 PM CDT> <Notice> <WebLogicServer> <BEA-000365> <Server stat
<Aug 13, 2006 7:24:57 PM CDT> <Notice> <Security> <BEA-090171> <Loading the ident
A-000331> <Started WebLogic Admin Server "examplesServer" for domain "wl_server"
<Aug 13, 2006 7:25:00 PM CDT> <Warning> <WorkManager> <BEA-002919> <Unable to fin
map to the default WorkManager for the application bea_wls9_async_response>
<Aug 13, 2006 7:25:00 PM CDT> <Warning> <EJB> <BEA-014014> <The message driven be
chPolicy" that refers to an unknown work manager. The default work manager will b
Could not invoke browser, command=netscape -remote openURL(http://192.168.1.98:70
can open from the cmd-line.
java.io.IOException: java.io.IOException: netscape: not found
<Aug 13, 2006 7:25:00 PM CDT> <Notice> <WebLogicServer> <BEA-000365> <Server stat
<Aug 13, 2006 7:25:00 PM CDT> <Notice> <WebLogicServer> <BEA-000360> <Server star
```

图 17-2 不可能浏览

图 17-3 来自 WebSphere 6.1。它和前一个例子使用相同的 `java.util.logging` API，但是它用一种更好的格式替代了糟糕的默认格式。这是一种人眼可以浏览的格式。一旦你知道 I、W 和 A 指示了不同的严重级别（“information”、“warning” 和 “audit”），那么浏览警告和错误就很简单了。空格分开的、分栏书写的格式能帮助人类阅读文件。注意消息代码字段，它有助于自动分析文件。这种日志格式对人类和计算机都有相当大的助益。

```
[8/14/06 8:22:14:653 CDT] 0000000a SSLComponentI I   CWPKI0001I: SSL service is
[8/14/06 8:22:14:813 CDT] 0000000a WSKeyStore   W   CWPKI0041W: One or more key
[8/14/06 8:22:14:848 CDT] 0000000a SSLConfigMana I   CWPKI0027I: Disabling defau
[8/14/06 8:22:24:639 CDT] 0000000a WorkSpaceMana A   WKSP0500I: Workspace config
[8/14/06 8:22:25:508 CDT] 0000000a FileRepositor A   ADMR0010I: Document cells/t
[8/14/06 8:22:25:961 CDT] 0000000a SSLDiagnostic I   CWPKI0014I: The SSL componer
[8/14/06 8:22:26:325 CDT] 0000000a FileRepositor A   ADMR0010I: Document cells/t
[8/14/06 8:22:26:670 CDT] 0000000a SSLComponentI I   CWPKI0002I: SSL service ini
```

图 17-3 一种有助于浏览的格式

图 17-4 显示了同样来自 WebSphere 启动顺序的输出，但是使用了 JDK 的 `java.util.logging` 包的默认格式。我不知道谁出的这种两行格式的馊主意，这种格式使得扫描日志几乎不可能。而且，两行格式也使其他程序扫描日志变得很困难。程序 `grep` 不知道如何处理这种两行格式。你不得不去寻觅个老派的 UNIX 黑客来做些 awk 技巧。这种格式让人和机器都没招了。

```
Aug 19, 2006 7:13:25 PM com.example.server.SSLComponentIdentity emit
INFO: SSL service is initializing the configuration
Aug 19, 2006 7:13:25 PM com.example.server.WSKeyStore emit
WARNING: One or more key stores are using the default password.
Aug 19, 2006 7:13:25 PM com.example.server.SSLConfigManager emit
INFO: Disabling default hostname verification for HTTPS URL connections.
Aug 19, 2006 7:13:25 PM com.example.server.WorkSpaceManager emit
INFO: Workspace configuration consistency check is false.
Aug 19, 2006 7:13:25 PM com.example.server.FileRepository emit
INFO: Document cells/trozNode01Cell/security.xml is modified.
Aug 19, 2006 7:13:25 PM com.example.server.SSLDiagnostic emit
INFO: The SSL component's FFDC Diagnostic Module com.ibm.ws.ssl.core.SSLDiagnosti
Aug 19, 2006 7:13:25 PM com.example.server.FileRepository emit
INFO: Document cells/trozNode01Cell/security.xml is modified.
Aug 19, 2006 7:13:25 PM com.example.server.SSLComponentIdentity emit
INFO: SSL service initialization completed successfully
```

图 17-4 JAVA.UTIL.LOGGING 默认格式

Voodoo Operations

就像我前面所讲的，人类非常善于发现模式。实际上，我们似乎对发现模式有一种天然的偏好，哪怕模式并不存在。在 *Why People Believe Weird Things* [She97]一书中，Michael Shermer 讨论了模式发现对于人类进化的影响。在早期人类中，相比那些发现不存在模式的人（比如遇到碰巧看起来像豹子的灌木丛，拔腿就跑的人），那些不能发现真实模式的人（比如发现不了斑驳丛林中豹子的形状）传递基因的可能性更低。

换句话说，误确认（发现不存在的模式）的代价是很小的，然而误否认（没有发现存在的模式）的代价是很高的。Shermer 声称这种进化压力创造了一种迷信的倾向。我真的遇见过这样奇葩的事情。

遇到一个濒临失败的系统，运营部门的管理员需要通过观察、分析、推测处理，然后快速行动。如果这个行动解决了问题，它就会成为知识的一部分，甚至是文档知识库的一部分。可是谁说这是正确的行动呢？万一它只是巧合呢？

在我早期的一个商业应用项目里，我见过运营团队的一次实践不比魔法好多少。我碰巧在某个管理员的工位上，这时她的呼机响了。一看到消息，她立刻登录到生产服务器，启动了一个故障备份数据库。我有点好奇，又有点担心，问她发生了什么事。她告诉我有条消息显示数据库服务器要宕机了，所以得切换到另一个节点上，并重启了主数据库。看到那条信息时，我打了个冷战。消息说："达到数据通道生命周期限制，需要重置。"

我当然认得那条消息，是我自己写的。实际上它跟数据库毫不相关，是个调试信息，告诉我连接到外部供应商的加密通道已经运行足够长的时间了，密钥可能容易被发现，因为通道提供的加密数据量已经很大了。这大约每周会发生一次。

一部分问题在于消息的措词。"需要重置"没有说需要谁去重置。如果看了代码，就会清楚应用程序自身将会在产生那条消息后重置通道——但是消息的用户看不到这个代码。还有，这是我留下的一条调试信息，让我能知道在正常流量下多长时间将发生一次。我只是忘了关掉它。

我追溯到大约 6 个月前的一次系统失败，就在系统发布后不久。“需要重置”消息是在 Sybase 服务器停止前的最后日志。没有因果关系，但是有时空联系（没有关于数据库崩溃的警告，这需要 Sybase 的补丁，我们已经在那次停运后打上了这个补丁）。这样的一个时空联系，再加上一条用词隐晦的模糊消息，导致管理员在六个月后，每周都会在系统使用的高峰期执行一次失效备援。

最后的提示

消息应该包含一个可以用来追溯事务执行步骤的标识符。它可以是用户 ID、会话 ID、事务 ID，甚至是请求到来时分配的一个任意数字。当需要阅读 10 000 行的日志文件时（比如停机后），有一个查找字符串将节省大量时间。

有意义的状态切换需要被记录下来，即便你计划使用 SNMP 陷阱或 JMX 通知来报告监控信息。对状态转换做日志需要多花几秒钟编码，但是它为下游增加了选项。此外，状态转换的记录在事故后验阶段也非常重要。

17.5 监控系统

即使是最佳的日志系统也只能在应用程序实际运行时才有所帮助。僵死进程不会有任何日志，更不用说挂起的进程。对于这些情况，进程自身之外的某些实体一定也在监控——一些黑盒工具监控应用程序和主机的健康和状态。

这是监控系统的范畴。图 17-5 非常简要地描绘了监控系统。第三方检查者的关键组件就是收集信息的代理、可靠的传输机制，以及信息的显示和处理。

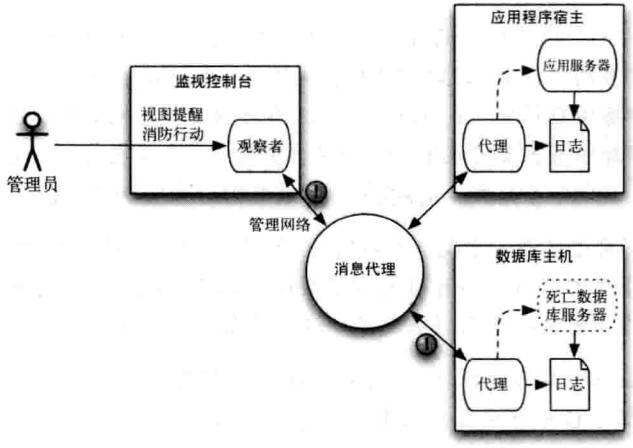


图 17-5 监视系统的概念视图

大部分的监控系统在被观察的主机上运行代理。这些代理定期观察一些情况，至少包括操作系统性能统计、进程健康状态、日志文件模式匹配，以及端口监听者。对于 Unix 主机，它们扫描 syslog 和管理员配置的任何其他日志文件。在 Windows 主机上，代理也扫描 Windows 事件日志。在本例中，数据库服务器上的代理检测到数据库停机。它向消息代理——作为监控系统自身的一部分而运行的嵌入式组件——发送事件。消息代理将该警报通知到所有的活动客户端。在本例中，管理员的监控控制台——访问监控系统的客户端应用程序——将警报显示为文字信息，并改变数据库图标的颜色。

这些代理很善于检测要寻找的事件：日志文件中的模式，SNMP 陷阱（很快将会涉及），停止的或者比正常情况下消耗资源更多的进程，快满的文件系统，等等。未曾预料的行为或者新行为可能检测不到，除非它触发了这些事件中的某一个。例如，如果日志文件中的所有错误都是用相同的格式报告，那么这些意外错误就能被捕获。另一方面，如果一种错误没有记日志或者使用了不同格式的日志，那么监控系统将不能捕获它。

这些代理就像是随军记者。它们离行动很近，也暗示代理本身是有风险的。当整个主机停机时，代理肯定也随之去了。因此，监控系统通常使用一些心跳机制来检查失效的代理——或者母船和代理之间的网络故障。

随军记者必须有某种方法来提交他们的报导，监控代理也是如此。监控系统严重依赖于它们可靠的消息传输机制。在中小型企业部署中，监控流量有时候像生产流量一样穿越相同的网络部分，这样做不好。一方面，这意味着生产网络的一个问题——比如网络蠕虫、DDoS 攻击，或者普通的老式配置错误——将自动禁止监控。监控流量经常包含敏感信息。虽然它不包含密码，但也经常有主机名、用户名、内部 IP 地址、日志文件的片段、进程名以及进程 ID，等等。所有这些都不能公开，因此它们不能穿越承载公共流量的 VLAN。

无代理监控

有些监控系统宣传“无代理”安装，其宣传点是去掉被监控主机上的代理，会降低 CPU 监控它们的开销。这本质上是一种营销策略。即使监控信息是从另一台主机上收集，这仍然需要与被监控主机上的操作系统合作。例如，微软操作管理器（MOM）能在有无数代理的模式下运行。此配置下，管理服务器用到了被监控主机的内置的事件日志和远程管理能力。因此，信息仍然在收集，也仍然消耗着被监控主机的资源。

商业监控系统

以前，只有大型企业对大型商用监控系统有兴趣。像 IBM、CA 和惠普等供应商只会追求与财富 500 强企业的百万美元的交易。随着整个商业世界已经切换到 24 × 7 的运行，系统越来越庞大。同时，供应商们在新的市场发现了机会。2005 年的卡特里娜飓风和 2001 年的 9 · 11 恐怖袭击，激发了对商业持续运转和灾难恢复的迫切需求。（在卡特里娜飓风之前，许多 IT 运营管理者已经在努力鸣响这种警报，但是灾难恢复计划对 CEO 和董事会从来都没有吸引力，直到他们看

到公司由于缺乏相关计划导致生意在减少。）萨班斯–奥克斯利法案第 404 节[1]的强制实施所创造的严格的管制环境，也使得安全控制和监控比以前更重要。这些力量的汇聚意味着现在商业监控系统成为了开展业务必需的一部分。

商业系统的不足

大供应商的工具集是稳定的、成熟的，受到良好的支持。用《跨越鸿沟》[Moo91]一书的说法，现在这些系统甚至已经穿透了“最后的干流”——绝对是最难卖这些技术的市场。如果它们有缺陷，那就是在它们朝向 IT 的那一端。将生产系统的运营从商务结果中分离出来的做法现在不再被接受——这从来都不是个好主意。

企业级应用管理

总体而言，这些系统致力于网络和服务器管理。一批致力于应用程序的小公司在 90 年代末期快速成长。不像它们前寒武纪的表亲，这些深思熟虑的应用程序的可用性和性能首先取决于用户体验。大鱼吃小鱼，这些小公司已经从市场上消失了。大供应商——HP、EMC、BMC、IBM、CA（除非你足够大，能以两三个字母简称被知晓，否则你就不能参与这一行的游戏）——正在集成这些关注应用程序的产品到它们的旗舰套装中，创建新兴的 EAM（企业级应用管理）。

EAM 套装集成了网络、服务器以及应用程序能力（理论上），为你的系统提供内部的完全可见性。通过嗅探网络数据包和自动执行基线重设，它们可以自动发现依赖性。EAM 保守其承诺，并且可能交付伟大的事情，特别是开始进一步容纳像配置管理、自动部署以及负载和压力测试等应用程序生命周期功能。这部分市场正在快速发展。

把运营和商业结果连接在一起需要使“系统”信息和“商业”信息相关联。例如，在当今一个典型的分布式系统中，你能一直识别出哪个服务器提供特定的功能吗？可能不能。更可能的是，一个功能由一群跨越多层的服务器提供。因此，监控系统不仅要意识到系统，还要意识到这些系统提供的商业功能。实际上，任何时候出现系统事件时，都应该能够识别出对这些功能的影响——无论该事件是一个问题还是偏离正常值的度量信息。直到现在，多数商业系统都假设应用程序不提供自身的任何可见性。它们已经开始贯彻这类意识，但是还有很长的路要走。

如果说这系统还有另一个重大不足，那就是它们只能告诉你系统认为正在发生的事情。让环境中的每个模块自行启动和运行是完全可能的，但是对终端用户依然有坏结果。这经常在阻塞线程或者级联失败事件时发生。甚至即使大多数用户被正确服务，因为信息、浏览器、cookie 或者

① 我发现 404 节有些要求很搞笑，比如要求对敏感信息的控制，以及遇上 404 错误（找不到网页）时，要能验证每个财务信息片段的来源。不过我要警告说，在我参加过多数聚会里，这个笑话已经没有“笑”果了。

其他外部因素问题，一些用户也还是可能有问题。目前，主要系统都还没有体现单独用户的视角，它们只是表现系统自身的视角。有一些新产品（暂时还没被大鱼们吞下）把自己放到了用户的视角上。

环境知觉

在《全息显示：从架构空间转换为人与数字信息之间的接口》[WID+98]一书中，MIT 多媒体实验室的一些研究员支持全息显示。这包括惟妙惟肖的声音、光线和像水中的涟漪那样的运动模式，每个都映射到一些数据中。例如，一个纸风车可能映射成网站上的进入流量，声音可能表示系统的内在健康。

当一切正常时，居民的声音感知逐渐融入背景中，就像夏天夜里的蟋蟀叫声融入到背景中。人类大脑伟大的模式识别系统可以抑制不变的输入，至少在没有威胁时。这样在环境中待了一两天后，居民将明白“正常声音”是什么样子，就像船上的工程师那样。

当下层数据改变时，感知环境的数据会改变它的声音或模式。想象那些蟋蟀突然一下子安静下来。这些蟋蟀鸣叫声可能融入背景中，但是并不会完全消失。同一水平上，感知还在，突然的安静就仿佛是令人震惊的吵闹。

不是每个人都能生活在MIT多媒体实验室，但是有一个开源项目提供声音的环境知觉。Peep[①]能够监控网络中的状态和事件，并把它们转换为小溪和鸟的悦耳声音。那个项目4年来都没有发布更新，看起来是由于被忽略而失去了活力，等待着其他人接手。

在实用项目自动化（Pragmatic Project Automation）[Cla04]网站有很多关于监控系统（特别是构建服务器）健康的物理设备的文章。参见http://www.pragmaticautomation.com/cgibin/pragauto.cgi/Monitor/Devices。在上一个项目，我的团队从该网站上获得了灵感。我们接通了一个顶灯来监控我们的构建服务器。“灯是绿色时，构建就正常。”

① 参见http://peep.sourceforge.net/。

监控系统设计

在大多数情况下，监控系统已经为你选择好了。这些是要求广泛部署来体现价值的昂贵设备，一旦选择和部署，系统将会深入 IT 运营多年。这个决定影响深远，比公司采用什么操作系统、编程语言、硬件供应商和组织架构图要更持久。换句话说，在你设计一个新系统时就能同时使用监控系统的可能性是非常小的。因此监控系统也是你要设计的环境的一部分。

在这种环境中，很容易为特定的监控系统内建支持，导致紧耦合和绑定厂商的结果。你可以根据一小套官方或事实标准来设计应用程序，从而避免这种陷阱。

17.6 法律上及事实上的标准

监控系统的世界由一系列标准统治着。奇怪的是，这些标准貌似不能提供你想要的那么多的互通能力（interoperability）。让我们来看 SNMP，这是重量级的监控标准，以及一些后来居上的新标准。

简单网络管理协议

监控和系统管理的标准的老祖宗无疑是简单网络管理协议（SNMP）。SNMP 可回溯至 1988 年的 SNMP 第一版。虽然它最初是特别为管理网络设备制定的（所以名字中有“网络”），但是现在 SNMP 已经被扩展，远远超出了最初的范围，从承担互联网流量半壁江山的思科电信级路由器，到 USB 外围设备，都会用到 SNMP。

即使是第一版，SNMP 中的“简单”实际上也有点忽悠人。SNMP 从概念上讲很优雅，让它显得简单，就像我们说 LISP 是简单的一样。它只是把一个思想萃取出它最纯正的本质，然后应用到任何可能的问题上去。你只需将概念内化，深入地理解和体会，剩下的就变得清晰而简单了。在你能体会它之前，它看起来很乱。SNMP 的本质概念是“任何事情都是个变量”。一个节点能汇报或能做的任何事情都是个变量。没有命令，只有变量赋值。

SNMP 向设备发送一个“设置指令”，将变量设置为指定的状态，而不是告诉设备去做某件事情。例如，使用 SNMP 告诉 Packeteer 流量整形器一个新的 trap 接收器，看起来像是写入一个名叫 1.3.6.1.4.1.2334.2.1.6.2.0 的变量，也被称为 `trapDestAdd`。（你必须是标准委员会的一员才能提出像 1.3.6.1.4.1.2334.2.1.6.2.0 这样的名字。）

尽管 SNMP 自称为协议，但是整个标准不止定义了一个网络协议。定义 SNMP 的 RFC 也定义了一个信息模型，和一种如何在信息模型定义新组件的元标签。

信息模型被称为管理信息结构，它定义了 MIB（管理信息基础）。互联网编号管理局（IANA）[①]维护着组件拥有者的一张大表，网址是http://www.iana.org/assignments/smi-numbers。在另一种系统命名法下，组件定义通常也称为 MIB。供应商提供 MIB 来描述他们设备和软件的管理接口。以下是 Packeteer 定义的 MIB 的一部分：[②]

code/transparency/packeteer.mib

```
trapDestAdd OBJECT-TYPE
   SYNTAX DisplayString
   ACCESS write-only
   STATUS mandatory
   DESCRIPTION
```

① 参见http://www.iana.org。

② 来自http://www.packeteer.com/support/resources/utilities/mibs/8.0/packeteer.mib。

```
        "A shortcut for adding a host to trapDestTable. If the name is not
        in n.n.n.n IP address form, then the agent attempts to look it up
        via DNS. If the operation fails with BADVALUE, try again with
        an IP address. The associated
        IP address is added to the trapDestTable if the operation succeeded.
        The table should be queried afterward to
        insure that the action took place. "
    ::= { psAdmin 2}
```

如你所见，MIB 使用一种可移植的语言来定义。[1]给定一个 MIB，监控系统知道如何与设备或者应用程序交流。（为了导入到监控系统，MIB 通常需要编译成一个私有格式。）有了 MIB 的表、类型和变量定义，监控系统可以根据被监控系统的当前状态定义阈值、触发器以及警报。SNMP 还允许“代理”——服务器上处理 SNMP 自身的精灵进程或者库——在有关事件或警报事件发生时异步通知监控系统。这些警报被称为陷阱（trap）。陷阱比变量更通用，它们主要用来取得监控系统的注意。

很多软件厂商都采用了 SNMP。微软有一个对 Windows 可用的 SNMP 服务，允许任何安装了 Windows 2000（或更新版本）的计算机公开自己（好像它们当前还不够公开）。Apache 有一个 SNMP 模块，它被编译进 WebSphere 和 WebLogic。Oracle 支持几种 MIB，包括标准的和私有的。

所以，所有这些对你的系统意味着什么呢？使用这些平台中的任何一个都自动带来了对 SNMP 的高度支持，因此，直接透明化。使用基于 SNMP 的监控系统和支持 SNMP 的平台提供了上千个变量的直接公开化。监控系统还允许你为披露即时行为和异常状态的警报定义阈值和策略。

然而，对于你的应用程序代码，前景远不乐观。麻烦在于为定制程序创建 MIB 是一项极为特别的艰巨任务。就算是你自己写的 MIB，监控系统的管理员也经常不愿意安装来自“不信任”团体的 MIB。（我不知道为什么一个开发组织可以被信任去开发要承担数百万美元营收的应用程序，却不被信任去创建一个监控系统的 MIB 插件。总之，我经常看到管理员努力防止这么做。）

除此之外，创建 MIB 之后，你必须有一个嵌入到你的应用程序的 SNMP 代理，它是 SNMP 变量和你的应用程序中的对象之间的桥梁，如图 17-6 所示。（尽管标准将 161 定义为代理的标准端口，但还是可以运行多个 SNMP 代理，使用各自的端口。）虽然 SNMP 将 MIB 变量作为对象，但它们其实更像 C 语言的 `struct` 结构。一个 MIB 本质上定义了一大块全局变量——一些是数组，一些是结构体 `struct`——没有必要都映射为对象。

① ASN.1——如果你感兴趣的话。参见http://www.asn1.org/。

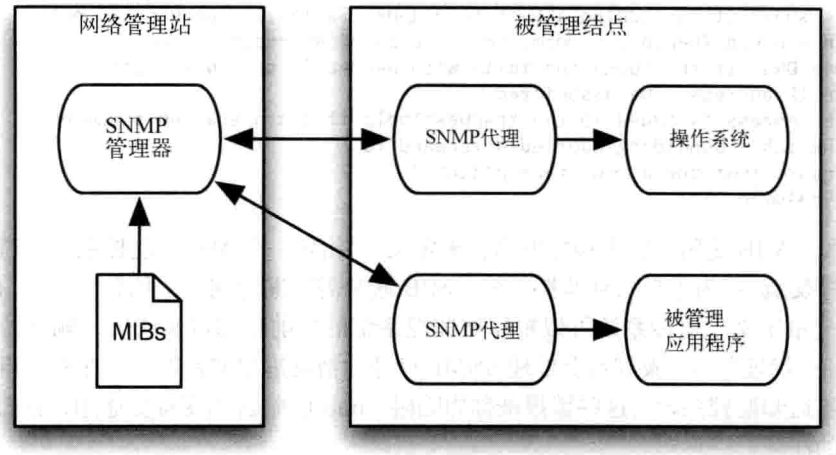


图 17-6　SNMP 通信结构

在 Java 社区，通过写应用程序代码支持 Java 管理扩展和使用 JMX-to-SNMP 连接器，就可以很好地支持 SNMP 兼容性。

CIM

1996 年，分布式管理任务组（DMTF）[①]引入了一个 SNMP 的继承者和竞争者，称为公共信息模型（CIM），这个模型使用一种元对象协议和允许动态注册和发现被管理资源的代理结构，替代了笨拙的 ASN.1 和 MIB 结构。

在很多方面，CIM 在技术上比 SNMP 更先进。CIM 的面向对象模型更贴近当今的编程风格，它的动态发现能力意味着管理平台不再需要进行“一揽子”的大采购。

目前，CIM 远不及 SNMP 普及。大多数操作系统厂商支持 CIM，包括微软、Sun、HP 和 IBM。Linux 支持已存在但是没有广泛集成到发布包中。许多硬件厂商在他们的企业级产品线支持 CIM。有些应用厂商支持 CIM，比如 Oracle。然而应用级的支持远远没有普及，所以我认为 CIM 是一个远景，今天不用太多讨论。

JMX

Java 管理扩展（JMX）历史上曾经是最被忽视和未受到赏识的 Java API 之一。在 1998 年作为 JSR 3 被引进，最初期望它仅作为 Java 企业版的一部分。随着时间流逝，JMX 已经充分证明了自己的价值，它甚至成为了 JDK 1.5 标准版本的一部分。任何运行 Java 5 或更新版本的 JVM 将

① 见http://www.dmtf.org。

自动开启 JMX。

SNMP 使用全局变量、结构体和表，看起来更像过程式语言，而 JMX 则展示了管理的面向对象的视角。管理的根本单元是一个 MBean，一些底层操作系统对象的“管理代理”，如图 17-7 所示。MBean 的公共属性——在 JavaBean 世界的意义上——和方法可以远程调用。

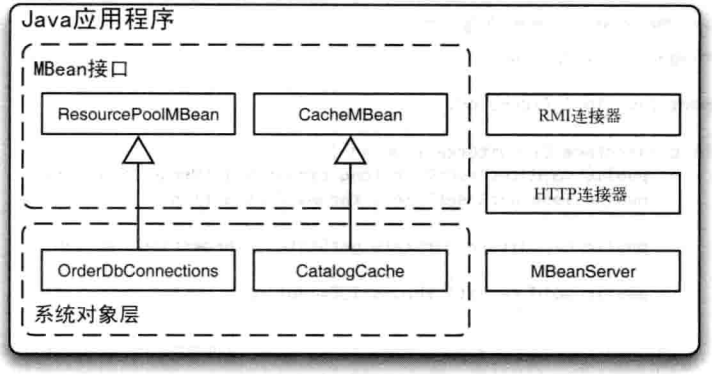


图 17-7 作为代理的 MBeans

JMX 发狂了

和大多数应用开发者所需要的相比，完整的 JMX 规范所允许的复杂行为要多得多。MBean 可以被动态地实例化，这可能会激活全部的子系统。域对象可以有自己的 MBean，它们动态地弹出或消失。模型 MBean 是如此动态，编译的时候它们甚至还不存在——它们通过编程来定义接口，并附加对象，从而被装配。

就创建透明的应用程序而言，MBeanServer连接到应用程序架构的长寿命组件：资源池、缓存、存储库和连接外部系统的接口，等等。

这些动态的，采用了最新技术的特性的确对平台开发者有益——比如 JBoss 的开源工作组，和 IBM 忠诚的 WebSphere 开发者。但这些特性对应用程序开发者几乎没有价值。奇怪的是，如果你深入到了 JMX，就会陷入迷茫了。暂停打磨 JMX 的“尾翼”并备份，呼吸下清新的空气，然后去写个故事卡吧。

MBean 有名字，注册到一个 MBeanServer。它位于允许远程访问的外部连接器和内部 MBean 之间。外部连接器采用像 HTTP 和 RMI 协议自行调用 MBeanServer。（也可能以被管理资源的身份从同一个 JVM 中调用 MBeanServer。应用程序不应该这么做，这完全是绕圈子的自言自语。）使用 RMI 连接器，可以取得代表被管理 bean 的远程代理。在远程代理上调用的方法将在被管理 bean 自身上调用。这些可以是简单的属性存取器或者控制方法。

图 17-8 显示了一个断路器组件的简单管理接口。方法 setResetTime(long timestamp)和 getResetTime()定义了一个 JavaBean 风格的属性。方法 reset()强制被管理对象的一个状态转移。这只是接口，因此必须有一些类实现它。确实，被管理的对象通常实现了这些接口。图 17-9 显示了一个被管理资源的例子。这意味着 JMX 的直接使用使得代码“知悉”自己的未来，它将是一个被紧密观察的应用程序。

code/transparency/CircuitBreakerMBean.java

```
package com.example.util;

import java.io.IOException;

public interface CircuitBreakerMBean {
        public void setResetTime(long timestamp) throws IOException;
        public long getResetTime() throws IOException;

        public CircuitBreakerState getState() throws IOException;

        public void reset() throws IOException;
}
```

transparency/CircuitBreakerMBean.java

图 17-8 MBean 的简单例子

code/transparency/CircuitBreaker.java

```
package com.example.util;

public class CircuitBreaker implements CircuitBreakerMBean {
        private CircuitBreakerState state = CircuitBreakerState.CLOSED;
        private long resetTime = -1;

        public void setResetTime(long timestamp) {
                this.resetTime = timestamp;
                checkForReset();
        }

        public long getResetTime() {
                return resetTime;
        }

        public CircuitBreakerState getState() {
                return state;
        }

        public void reset() {
                setState(CircuitBreakerState.CLOSED);
        }

        ...
}
```

transparency/CircuitBreaker.java

图 17-9 MBean 的接口实现

另一种 MBean——动态 MBean——不需要编译时信息。类似于 CORBA 的动态调用接口或者

Java 自己的反射 API，动态 MBean 描述它们自己的方法，而不是静态的、在编译时定义插入桩。这意味着，一个足够通用的动态 MBean 应该能够为任何旧 Java 对象扮演管理代理。确实如此，StandardMBean 恰恰做了这些。给定一个对象和接口类，它成为一个 MBean，暴露对象的接口。这绝对是向你的应用程序添加 JMX 支持的最快方法，并且还有从系统对象自身中移除了 JMX 污染的额外好处。

Java 5（JDK 1.5）之前，JMX 是一个扩展。为了使用它，应用程序必须包含一些 JMX 规范的实现。最常见的是 Sun 和开源 MX4J 项目的 JMX 参考实现。自从 Java 5 以后，JMX 被集成到了 JVM，所以任何应用都可以使用 MBeanServer。

尽管依然很少被应用程序开发者使用，JMX 已经在平台开发者中取得了广泛支持。JBoss 内核已经收缩成一个仅仅基于 JMX 的注册机和模块加载器。IBM WebSphere 使用 JMX 来公开自己。BEA WebLogic 在中间层某个位置，WebLogic 的关键部分是 JMX 激活和管理的，而不是 JBoss。自从版本 1.2，Spring 框架已经提供了一个自动的动态 MBean 生产器。在典型的 Spring 风格中，需要一组古怪的 XML 而没有代码。（好吧，XML 最终是清楚的，但是就像一个模糊的预言，它仅在回顾时才是清楚的。）

这些平台上的 JMX 支持带来了最好的益处之一：脚本化能力。BEA 和 IBM 都创建了可以通过远程连接器访问 MBean 的命令行 shell。BEA 的 wsadmin.sh 和 IBM 的 wslt 都为启用 MBean 的应用程序服务器提供了脚本 shell 接口。（见 14.4 节“管理接口”中对图形化管理接口的讨论。）JBoss 拥有“twiddle”，它不是一个完整的 shell，但是允许在任一 bean 上调用命令。

> 很难从运营部门的人得到赞赏，他们是很难打交道的。

但是，让你的应用程序管理功能脚本化，他们会念你的好的。我怎么夸大脚本化管理接口的价值也不过分。通过 JMX 提供有用的 MBean 集合，是你能为 Java 应用程序提供脚本化管理的最容易的方式。

公开什么

如果你能预测哪些度量信息将限制性能、揭示稳定性问题，或者暴露系统中的其他缺陷，那么你可以只监控这些信息。做这种预测会有两个问题。首先，你可能会猜错；其次，即使你猜对了，关键信息也在随时变化，代码和需求模式都在改变，一年前的关键点现在甚至可能不存在了。

在应用程序内部，最理想的就是暴露每一个状态变量、计数器和度量信息。因为你不知道将来需要什么，干脆就公开所有的信息。因为你不能预测阈值应该是什么，或者违反阈值时该如何响应，所以你应该始终坚持提供可见性。把策略留在后期。现在提供完全可见性，但是将策略外在化，所以你可以推迟做那些决定。

当然，你可以为任何东西花费无比多的力气来创建 MBeans 或 SNMP 变量。由于你的系统除了收集数据还需要做些别的，我找到一些启发式的方法来帮助确定需要暴露哪些变量。有一些能立即提供，而其他的，你不得不首先添加些代码来收集数据。下面这些是我一直觉得有用的。

- 流量指标

总页面请求、页请求、交易数量、并发会话。

- 资源池健康

启用的状态、总资源[①]、签出资源、高水位标记、创建的资源数、销毁的资源数、检出的次数、等待资源阻塞的线程数、线程阻塞等待的次数。

- 数据库连接的健康

SQLException 抛出数、查询次数、查询平均响应时间。

- 集成点健康

断路器状态、超时次数、请求次数、平均响应时间、良好响应次数、网络错误次数、协议错误次数、应用程序错误次数、远程端点的真实 IP 地址、当前并发请求数、并发请求的峰值。

- 高速缓存健康

缓存条目、缓存内存使用、缓存命中率、垃圾收集器刷新的条目、配置的上限、创造条目所花时间。

所有的计数器都有隐含的时间组件。只要它们以“在最后 *n* 分钟”或“自从上次复位”结束，你就应该读取它们。

JMX和SNMP

JMX 用来支持不同的连接器和协议适配器。似乎通过 JMX 来为 Java 应用程序开放 SNMP 是很自然的。确实，有许多 JMX 到 SNMP 的连接器可以用，质量参差不齐。AdventNet（http://www.adventnet.com）似乎是这个领域的领跑者。

这些连接器需要跨越的最大障碍并不是衔接这些协议。在 SNMP 和 JMX 之间映射协议操作实在不算什么。毕竟，SNMP 只有三种操作类型：获取变量、设置变量，和枚举变量。

回想一下，SNMP 的信息模型是基于变量、表和结构的模式。JMX 是面向对象的。JMX 到 SNMP 的连接器将这种不匹配桥接起来。它必须决定哪种变量映射到 MBeans 的属性上去，哪个

① 这适用于连接池、工作线程池和其他的资源池。

变量会触发方法调用，如何将对象集合转化为表中的条目。最后，JMX 到 SNMP 的连接器也必须将 JMX 的通知（异步事件）映射到 SNMP 陷阱。

结果是一个完全配置的 SNMP 代理，这个代理使用 JVM 中的 `MBeanServer`，和一个 MIB——用来导入到监控系统中去。一旦障碍清楚，将 JMX 启用的 Java 应用程序集成到一个基于 SNMP 的监测系统就易如反掌了。

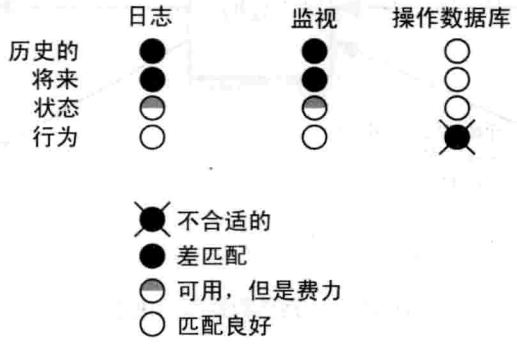


图 17-10 适应于透明技术的目的

17.7 操作数据库

日志和检测都有利于暴露和理解程序或系统的即时行为，二者在提供历史和未来视角方面都不是特别出色。图 17-10 从 4 个角度评判了这些方法。虽说有可能只通过日志文件来推导系统的当前状态，但是这很难，因为它需要回溯每个相关状态变量的上一个状态跳转。（当然，周期性的分离状态变量会有助于这一分析。）

监控系统善于表现系统的瞬间行为。它们能很好地表现当前状态，但是正如我在 17.5 节中所说的，它们只能表现这一块。当涉及到历史和未来趋势时，它们就力不从心了。在企业应用管理套件成熟之前，我不得不考虑这一差距。

图 17-11 介绍了一种互补技术，它作用于系统的面向时间的视图。这种“运营数据库”[①]（OpsDB）积累了来自所有服务器、应用程序、批处理任务和组成扩展系统的数据的状态和变量。它提供了一个“单一窗口”，这个窗口能表现仪表板、系统统计和它们之间关联性的业务指标。日志提供的是单个应用程序的可见性，而 OpsDB 的这种更广阔的视野则统一了整个系统的状态的度量报告。

OpsDB 作为日益流行的仪表板的数据源，标识了当前的状态群。因为它包含需求指标和系统度量，一些数据挖掘中的操作会表现容量规划所需的相关要素。OpsDB 中包含的历史记录也允许自动基线来确定整个站点的“正常”指标是什么样的。

① 我喜欢称它为“操作数据商店”，但是这个名字已经用在其他地方了。

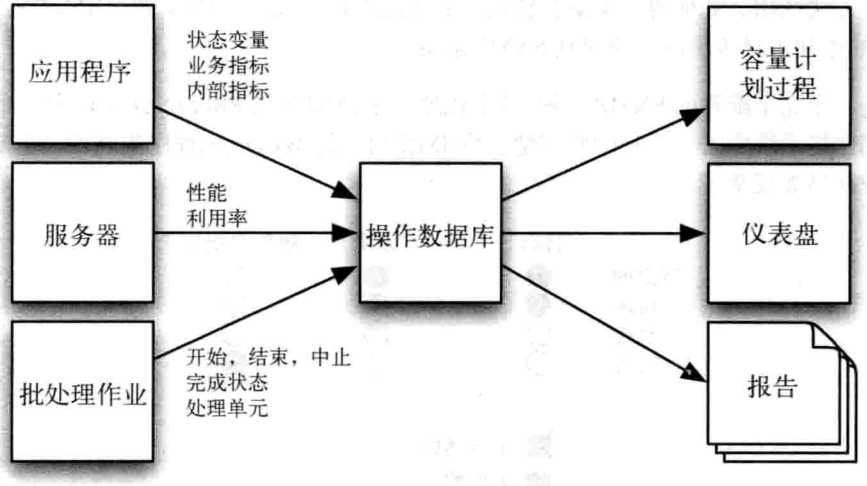


图 17-11 运营数据库的角色

当应用程序级的问题发生时（比如来自一个通道的过时数据），作业执行记录能让排错更快。检查正常的启动和结束时间，将它们与处理的项目数量关联，也能说明我们应该期待作业何时完成，以及它们之一是否有出问题的危险。

OpsDB高层结构

图 17-12 所示的 UML 类图，表现了 OpsDB 中的信息。[①]将这些对象模型映射到 SQL 模式是行使的一个读者（不分，是因为这种映射的形式应该有特定的 ORM 工具和数据库服务器来驱动）。

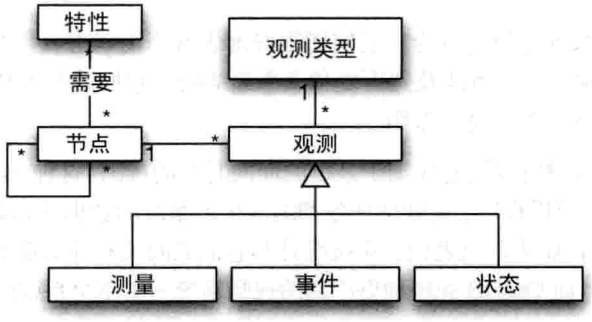


图 17-12 OpsDB 对象模型：观测

① Martin Fowler 的《分析模式》[Fow96]的读者会认识到这一点类似于观察者模式。这种模式描述了一种访问病人健康或疾病的医疗诊断系统。这个概念直接转化了。在这里，不是跟病人打交道，你是在处理联网的计算机系统——一个复杂度低得多的东西。

特性表现了一个业务上比较重要的功能单元。这些和系统可用性 SLA 所书写的功能相同，它们需要容量规划来度量。

特性并不来自于某一台服务器。一个特性很可能需要至少 3 个主机（网络服务器、应用服务器和数据库服务器）、每一层的许多应用程序、一堆防火墙、一个网络交换机，和一个存储阵列或 SAN 才能实现。节点类代表了这些活动节点。实践中，它足以代表主机和应用程序了。如果系统包括硬件防火墙、SSL 加速器、内容缓存服务器，或其他的扮演活动角色的网络设备（主要是关于第 7 层交互的），那么它可能要包括这些节点。每个节点都有它独特的 ID。单个服务器、应用程序和批处理任务在报告它们的观测时使用这个 ID，因此仔细地跟踪它们很重要。（我发现，给特别的组或分组分配块 ID 效果很好。这保持节点 ID 的连续，并让他们可以用完节点 ID 而不用等待新的分配。）节点再使用其他的节点，但是捕获这些依赖可能很麻烦，因此这只是可选的方案，不强制使用。如果要使用它们，节点依赖能帮助排除故障。这些依赖也正是级联故障传播的途径。

观测是 OpsDB 的心脏。每个观测值都是从节点收集的一个数据点。对于服务器节点，观测主要是性能统计，它作为测量指标被记录。测量是周期性的，所有的记录都会保存。对于应用程序来说，这些将包括性能统计数据和重要系统对象的状态。例如，应用服务器应该记录它的数据库连接池的高峰和签出资源（当然还有其他事项），但是它也应该记录从“启用”到“禁用”的状态变化。同样，断路器应该记录每次状态跳转。状态跳转作为状态对象记录下来。对于当前状态的仪表板，只关心上次状态条目。而对于故障排除或历史趋势，状态改变的频率和类型常常很重要。

每个观测都需要一个观测类型。全体观测类型定义了 OpsDB 中的整体信息。观测类型定义了观测的名字和具体的子类型。

喂养数据库

我建议创建一个客户端的 API 来给 OpsDB 输入观测数据。它应该使用大多数系统所使用的语言。首要的是，要保持简单。一切都可能失效，包括这个新数据库。它不是系统财务上成功的关键，因此 OpsDB 的失效不应该对系统主功能上有重要的影响。

如果系统包括 shell 脚本或批处理文件，它们将需要调用一些命令行工具来写 OpsDB。

脚本应该在启动时调用命令行，用来记录有多少条目需要记录。完成时，它会记录实际处理了多少条目（而不是有问题的条目数）。如果工作完全失败，它应该记录非正常终止。

如果是 Java 系统，你又想做得更多些，可以写个通用的 `MBean` 记下周期性的样本。它甚至还可以收到状态变更的通知并记录下来。然后，使用任何应用就只要简单地实例化和配置 `MBean`。

使用操作数据库

一旦 OpsDB 的数据体建立了，它就变得更加有用。图 17-13 在图 17-12 的模型上增加了新的

类。（“特性”和“节点”都从图中省略了，但它们还存在。）

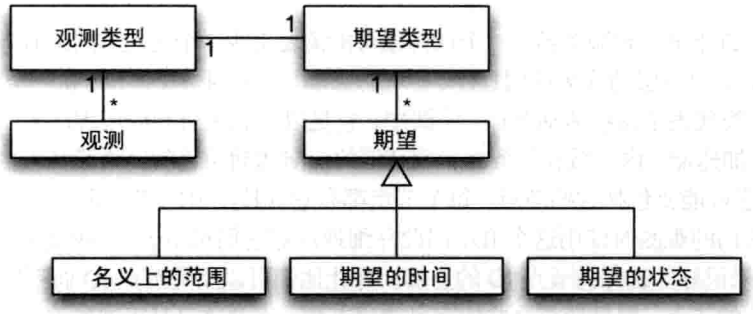


图 17-13 OpsDB 期望

一个期望类型对应于一个观测类型。它定义了期望的名称和特点。期望代表了度量允许的范围、事件会（或不会）发生的时间帧，或者允许的状态。违反了这些期望会触发监控系统的报警。

这些期望的最好数据来源是已经包含在数据库中的历史数据。期望应该与现实相符，因此你能避免这些误报的负面影响。

在初期，期望可能有所松动。随着时间的推移，流程控制得更好了，这些期望可以收紧，也能更复杂。例如，早期的对于“网络服务器 CPU 利用率”的期望认为“在 0%和 80%之间”是可接受的，随着你从系统关联里了解更多，期望可能收紧为“5%到 50%之间”。在下一个成熟级别，商业节奏可以加入了，要么作为连续的封装要么作为一个阶梯函数。假使作为一个阶梯函数，期望可能在晚间要求低 CPU 占用率，而在早上是中等，在午后是高。高于或低于预期的偏差都会触发警报。

随着时间推移，OpsDB 让系统更加成熟和内省。它将更好地认识自己，知道对外界刺激该如何反应。

对于大型系统，OpsDB 随着时间能累积大量数据。不要忘了“稳定状态”这个稳定性模式（参见 5.4 节“稳定状态”）。如果你不压缩数据，你的系统最终会在收集数据中陷入瘫痪。很明显，因为保持太多的古老性能数据而威胁到系统并不好。（多久算古老？这取决于你的系统，但是我认为按分钟采样一个星期以前的数据并没什么用处。）

误报的危险

在 *Inviting Disaster* [Chi01]一书中，James Chiles 提到一个工厂主管的故事，这个主管看到一个控制室的操作员本能地忽略了一个警告。当面的时候，操作员极力否认他这样做了，这并非因为他担心后果，而是因为他完全不记得自己忽略过这个警告了。系统训练了他完全忽略这种警告，以至于他这样做了却没有意识到。

我经常听到开发人员或经理佩戴传呼机。一位妇女告诉我她的呼机每天夜晚要呼叫三次。她知道，这表示正常。如果第三次呼叫在夜晚的某个时候没有来，她就知道出问题了。如果第四次呼叫来了，她知道也有问题了。这是一种情境意识的形成，虽然聊胜于无，但是我不能推荐大家选择这样一种生活方式。

17.8 支持流程

你曾经多少次遇到这样的情况？——报表系统生成一份报告，发送给列表里的人，而有一半的人在 Outlook 中设置了一条规则自动删除该报告。很经典。当然，遇到这种情况，我立刻就知道这份报告是没有用的。实际上，可能比无用更糟糕。不仅因为生成报告需要花费时间和精力，而且在底层系统变动时还得有人去维护报告。更糟糕的是，报告可能破天荒地显示出一次严重问题，但是报告的所有读者早就停止阅读它了。

报告是一个没有闭环反馈流程的透明度的例子。花费了金钱去实现和运营，却没有产生任何价值。它创造了一种虚假的安全感。如果没有人看，世界上最好的数据也毫无用处。本节中，我会暂时放下系统本身，去讨论更大的动态系统：创造和运营这些计算机系统的组织。

有效的反馈流程可以描述为“对有意义的数据反应灵敏”。系统中的透明度仅仅提供了访问数据的能力。环路中的人仍然需要查看和解释那些信息。

不同的流程学派有不同类型的反馈回路，像“计划–执行–检查–处理”的戴明循环，[①]或者约翰·博依德的非线性 OODA 循环。[②]最终，这些都归结到下面所示的每条上。

- 检查系统：当前状态、历史模式和未来预测。
- 解释数据。这经常发生在人们构思系统的模型时。
- 评估潜在行为，包括每种行为的代价，也包括不采取任何行动的代价。
- 决定行动过程。
- 实现选择的行动过程。
- 观察系统的最新状态。

① 参见 *Deming and Goldratt* [LC99]。

② 参见 *Certain to Win* [Ric04]。

约翰·博依德上校和 OODA 循环

OODA 反馈流程直接来自于约翰·博依德上校的工作，他是个空军飞行员和战略家。博依德上校的关键思想是 OODA 循环，即“观察–定位–决定–行动”（Observe-Orient-Decide-Act）。博依德声称环境的完整知识是不可能得到的，即使是可以获取的，也很快就变了。他把战场环境看作是易变的甚至是无序变化的。在这个领域，胜利属于那些能最好地反应和控制多变的战略战术的军队。

OODA 循环要求正确地观察，不被自己的希望和偏见笼罩——这绝对是很苛刻的要求。定位是根据前一个意象图和新的观察得出一个新的意象图的过程。

好的定位确认什么是可能的和不可能的。在军队和公司中，通过策略过滤观察和定位是至关重要的。“杜撰”（Spin）和 OODA 是对立的，它要求参与到真实环境中，而不是关于环境的想象和理想画面。

OODA 包含很多反馈步骤。观察、决定和行动都影响到行动者的定位。在行动时，你不仅改变了自己关于演变场景的理解，也改变了外部环境。这样，一次 OODA 迭代在两个方面影响到下一次迭代：一是通过环境本身，一是通过你自己对环境的看法。

每次循环，你都比以前知道得更多，创造了正反馈。每次你看到正反馈循环时，应该想到系统是非线性和无序的。

无序能力将OODA和纯粹的刺激/响应反应区分开来，在刺激/响应下，刺激的增加导致响应也成比例地增加。试想一战中的战壕战，或者试图通过“做更多的流程”获得更快的软件。通过用更快的 OODA，你可以获得无限的创造力来瓦解你的竞争对手。

博依德称此为深入敌人的决策环：既预料又限制了他的行动。[①]

① 我还将 OODA 看作是敏捷开发方法的强大论据。每个开发迭代都应该映射成一个 OODA 迭代。导致的环境控制允许公司处处主动，瓦解竞争。Steve Adolph 的论文更深层地考察了 OODA 和敏捷之间的联系。http://www.steveadolph.com/articles\%20and\%20papers/Adolph\%202006\%20agile\%20lessons_final.pdf。

观察的关键

首先和最重要的是，观察者应该既观察趋势又观察异常值。两者都提供了有价值的信息。它有助于构建一个改善例行操作而不是间歇性努力的操作节律，下面是一些有助于观察的惯例做法。

- 每周，审查上周的问题单据。查找反复发生的以及那些耗费最多时间的问题。查找导致很多问题的子系统或者一个开发团队（如果有多个开发团队）。查找与特定第三方或者集成点相关的问题。

- 每月，查看所有问题。考虑问题类型的分布。随着严重问题被改正，总体趋势应该是严重程度递减的。还应该在量上是大致递减的。（随着新代码发布引入新问题，将会出现锯齿状的反复。）
- 每天或每周，查找日志文件中的异常和栈。联系这些找出异常最常出现的代码。考虑考虑这些是意味着严重问题，还是只是代码错误处理的缺陷。
- 回顾服务台遇到的常见问题。这不仅能指出用户接口的改进方向，还能指出系统在哪些地方需要更加健壮。
- 如果有太多的单据和服务台呼叫需要彻底审视，就寻找顶层分类，并随机抽取单据来找出让你头疼的事情。
- 每四到六个月，重新检查那些依然正确的原有相关性。
- 至少每个月，检查数据容量和查询统计。
- 查询最耗时查询的数据库服务器。这其中的查询方法有过修改吗？某个新的查询是否在最耗时列表中？这些修改中的任何一个都可能表明某处数据的堆积。最常见查询中是否有某些导致了表扫描？这可能表明丢失了一个索引。
- 查看每周和每月的需求（驱动性变量）和系统度量信息的包络线趋势。流量模式改变了吗？如果你碰巧看到流行时间的流行度在下降，这可能表明系统在这些时间太慢了。是否在驱动性的变量上表现平稳？这表明存在某些受限因素，可能是系统响应性。

记住，兴趣的焦点是会随着时间转移的。在早期，问题主要是反映式的。单据检查、事故事后报告以及最近趋势将是最有价值的。随着根本原因被改正，新代码发布，流量模式改变，重点将会从反映式过渡到预测式分析。下一季度会发生什么呢？明年的这个时候你需要在哪？随着这个重点变动，停止检查一些旧的事情，开始检查新的趋势。一旦某个度量信息停止产生有用的信息，就停止检查它。在发布后一个月里你所依赖的这个报告，在两年后将是没有价值的，甚至还会误导人。

对将检查的每个度量信息，考虑下面的事情。它相比历史正常行为怎么样（如果 OpsDB 有足够的数据可以开始产生预测，就很容易）？如果该度量信息继续着最近的趋势，其他相关的度量信息会发生什么？趋势能持续多久——什么限制因素会生效？限制因素将产生什么的结果？

例如，考虑“接收到的订单”和“应用程序服务器 CPU 利用率”之间的联系。在任何类型的零售网站中，这些度量信息将在某种程度上相关（“接收到的订单”是“独立访问数”的衍生，“接收到的订单”到“独立访问数”的系数就是营销人员所说的转化率）。因此，如果你看到“接收到的订单”在增长，你可以问：“它能继续增长多长时间？”回答这个问题应该激发大家进行解释和做出决定。决定可能是什么都不做，或者是增加资源、赶走顾客，或者是优化应用程序代码。这些决定都体现了反馈环路的成功结束。

17.9 总结

当遇到现实世界时，系统会发生各种奇怪的事情。异常和事件可以一直保持神秘色彩，也可以是你学到的有价值的教训。是在生产环境中随着时间改进的系统，还是停滞不前或者逐渐消亡的系统，主要区别就在于透明性。

管理系统很流行。现在 SNMP v3 统治了这个领域。CIM/WBEM 克服了 SNMP 的许多缺点，在不远的将来将会成为主要力量。对于 Java 开发者，JMX 提供了到 SNMP 或到 CIM/WBEM 世界的途径。

透明性要求访问计算机和软件的内部。公开这些内部信息是首要前提。接下来，需要一些收集和理解数据点的方法。这可以借助于商业监控系统和 OpsDB。最后，需要反馈流程，以便根据获得的知识采取行动。

下一章考察当透明度揭示出需要改变时应该做什么。就是立足长远，考察如何构建随着时间优雅改变的系统。

第18章 适　应

不管设想多么大胆，模式多么重大，上马的系统总是比可能的要差。新的系统有缺陷，几乎不能适应实际需求。它会继续这样，还是会逐渐符合创造者的设想？

系统真正诞生的那天，并不是设计和开发开始时，也不是项目构想时，而是启动投产时。这是开始，不是结束。假以时日，系统会成长和成熟。它会具备新的功能。它会少一些缺陷，也可能增加一些。它会成为它应该的样子，甚至更好，变成我们需要的样子。最重要的是，它会变。一个无法适应环境的系统必然会胎死腹中。

18.1 与时俱进

新做的系统必须做一些正确的事情，否则不会发布，它也可能做出设计师没有想到的事情。还有些功能虽然能工作，但并不是按照预期的那样进行，或者它比预料的要难用。本质上，系统无法完美地实现解决方案的设想。

新系统不会在使用中自动变好。大多数时候，系统痛苦地训练了用户。系统可能被改变，更好地实现了解决方案的设想，即便解决方案是随着时间一直在变的，因此只有通过刻意的行动才能消除沟壑。

在 *The Evolution of Useful Things* [Pet92] 一书中，Henry Petroski[①]认为，老格言“形式服从功能”是错的。他提出了设计演化规则，“形式服从失败”。也就是说，叉和回形针等常见物的设计，变化的动机来自于早期设计中不好的方面，而不是好的方面。即便是最不起眼的回形针，也不是一开始就长成现在这个模样的。每一次新的改进，都是试图纠正原来的缺陷。

在软件开发领域，每个新版本的动机都来自要么是新功能（填补空白），要么是缺陷（减少

① 我不得不倾佩作者，他能用 18 页的篇幅来阐述叉的出现。

肿块）。正如回形针的形状随着时间而演化，软件系统的架构和设计也在根据每次迭代中真实和感知到的故障，随时做出适应调整。

但是，适应是有代价的。任何改变系统的行为都有成本：设计、开发和测试，加上发布的成本。物理学家和化学家将此看作“激活能量”。如果改变的成本超过了填补空白或消除肿块的回报，那么理性的选择是不去做修改。回报的计算必须严格依据现金流。另一方面，价值可能在于保护已经投入给系统本身的投资。

现金流：钱流进流出，是每个企业的命脉。由于金钱的时间价值，现金流在将来会更加重要。

最后，一个简单的经济学问题，系统第一次发布之后要发生的成本占整个成本的 40%至 90%。这可能是“维护”，或者可能被认为是新的开发。无论哪种，发布 1.0 版以后，开发成本会持续出现。从设计和架构来说，这是一个被彻底认识的老问题。但是发布更改带来的困难和成本没有被深刻理解。

本章后面部分讨论大型和小型架构选择会如何影响系统的适应能力。这些都是激活能量的“实现成本”。激活能量需求的平衡是发布变更的成本，这是隐形的成本，可能会高到让人惊讶的程度。我们最终的目标是，让系统的变化，也就是填补空白和消除肿块，带来的效益超出其成本。

产生的能量高于启动所需的化学反应被称为放热。我们想要我们系统的改变是能赚钱的，产生的钱要超出创造它们的花费。

18.2 适应性的软件设计

适应性软件设计方面有大量的文献。有几十种数得出名字的方法让软件完美地实现其功能。很多更新的方法甚至引入了随时间而变的原则，归之为“敏捷方法”。本书不可能赘述所有这些。

大多数方法集中在创造正确的功能，或者让功能随着时间而变化。然而，软件设计还有一些其他方面，涉及在不打破产品运营的条件下系统的适应能力。应该将这些问题看作是一种“涂层”，站在你选用的任何实际方法之上来考虑它们。

依赖注入

起个好名字很重要。Martin Fowler 给一个相对常见的技术起了个名字，然后编程界就突然闪耀出“依赖注入”的框架。[①]（另见 AJAX。）依赖注入的原则很简单，组件应该通过接口而非实例进行交互。相反，其他一些代理应该从松耦合的组件中解放出应用程序。

做得好的话，依赖注入可以鼓励松耦合。某种程度上，它确实有用。它也利于单元测试。定义和使用接口是用依赖注入来达到灵活性的关键。通过接口来协调对象能够在不知不觉中调换接口。这种调换能够用新的功能来替换现有接口，而且替换能够用假冒对象来做单元测试。使用接

① 参见http://www.martinfowler.com/articles/injection.html。

口的依赖注入能保留你做本地变化的能力。

依赖注入

当 J2EE 开始流行时，很多的框架开发者开始创造诸如 Pico，Spring 和 Apache Avalon 这些东西。比起典型的 J2EE 应用程序，它们有显著不同的质量。用这些“轻量级”框架完成的应用程序重点在用小而独立的组件实现接口。这些组件并不直接实例化其他组件。相反，它们通过接口来调用其他服务，并且“假设”这个接口的实现会由容器提供。

起初，这被描述为控制反转，这是应用程序框架的一个常见特征。然而，控制反转应用于更通用的框架扩展中，通过子类或回调来插入应用程序代码，从而控制的是框架而不是应用程序。（框架调用到应用程序，这是区分“框架”和“类库”的特征反转。）

Martin Fowler 用“依赖注入”来描述将引用注入到组件中。容器在运行时根据配置文件或应用程序定义来连接组件。组件本身不需要确切地知道它们在谈论谁。依赖注入带来了高适应性的代码，它们易于做单元测试，随着需求的改变而易于“重新连接”。

对象设计

“良好的面向对象设计”能产生多么大的能量和激情啊，即便是在这“对象革命”开始了 20 年后的今天。不用进一步证明，我能断言确实存在良好的面向对象设计。那些已经相信的人不用证明就会同意我，那些否认它存在的在我提供论据之前绝不会动摇。

从适应的角度讲，最好遵循的规则是那些老的教条：松耦合和高内聚。这些规则最初应用到源模块。松耦合意味着“不要触碰另一个文件的全局变量”，高内聚意味着“文件的大多数子函数应该使用文件中的大多数全局变量”。现在，似乎这些规则要做一些更新了。

现代面向对象中耦合的定义，更多是与行为相关，而非与变量相关。每当一个类使用另一个类的行为时，它就耦合了那个行为。一个类需要另一个类的行为数量定义了类之间接口的“宽度”。

显然，一定程度的耦合是必需的。完全不调用其他对象的对象做不了什么事。因此，这时的“松耦合”是指什么呢？考虑一个类暴露的所有公有函数。看看其他类调用的这些方法的子集。一个完美内聚的类只有一个子集，它包含了所有的方法，这些方法被这个类的所有用户调用。这不太可能。比较可能的情况是，有一些方法子集。这个类的不同用户使用方法的不同子集。子集越不同，也就是多个子集发现的方法越少，后来改变方法行为也越容易。这样适应性更高。并非巧合的是，它也更容易创造无类接口到组、名称，并抽象出这些方法子集。

把这些方法归纳为子集，内聚则描述为对象的内部状态会多大程度地与这些方法子集相互影响。换句话说，如果一个方法集合只接触了这个对象状态的子集，但是并没有被对象状态的其他方面影响，那么这个对象就不是内聚的。这可能表示有另一个对象隐藏其中，等待被重构为独立的类。

耦合影响适应性多过内聚。高内聚的类希望更多的上下文有用。例如，考虑一个简单的代表二维点的几何类。不可能这个点要求任何外部上下文都起作用。不用应用程序启动其他关联或全局状态（比如单件或对象注册）它就能运行良好。它就像是一块砖，能以同样的方式用在任何你需要它堆砌的地方。

还要考虑一个过于聪明的领域对象，比如一个典型的 `Customer` 对象。它很可能与三四个其他类交互，比如 `Account`、`Address`、`CreditCard` 等。如果 `Customer` 也知道关于它自身持久性的任何信息，它就可能会和 `PersistenceManager`、`TableDataGateway`，或其他的对象交互。这些交互都是必要的，也是有问题的。没有它们，这个对象就毫无用处了；有了它们，它需要更多的子结构来表现预期的行为。它不像砖，更像拼图的方块。它有旋钮、插槽和接点，来与其他方块精确匹配。

对象簇可能共同紧密协作，类似于金属晶体。这些对象共同形成紧约束关系，正如晶体中的原子也是紧密结合的。在金属中，小晶体意味着更大的可塑性。可塑的金属越多，从压力中恢复得越好。大晶体鼓励裂纹的形成。软件中，大的“晶体”使其难以改变软件。当对象参与到多个协作模式中，它们连接两个晶体，形成更大粒度的结晶，进一步减少了软件的可塑性。

对于晶体紧密结合的传播区域并没有限制。极端情况下，晶体能扩展到应用程序的边界。这时，每个对象都适合它最适应的目的。它完全适合并最终与其他每个对象关联。这些结晶甚至可能和用巴洛克方式排序一样美丽。它们不允许改善，因为没有改进的可能，也因为每个的移动都需要移动其他所有对象。这往往是死亡结构。开发人员蹑手蹑脚地通过水晶宫，安静地诉说，并试图不触碰任何东西。

协作和设计模式

听起来好像我对协作持消极态度，其实并不是。实际上，说到对象设计，我就是一个行为主义者。协作是核心。但是，我强调协作应该设计为关注内聚，并减少软件的晶体粒度。当我看到一个对象参与到多个设计模式中，每一个都和不同的对象组协作，我看到了难以改变，甚至是不能改变的无底洞。

当需要链接不同的晶体，这些链接应该尽可能最少地跨边界。

顺便说一下啊，有很多新手在从经典书《设计模式》[GHJV95]里应用新发现的知识时走错了方向。这本书鼓励对象交互的更多协作和行为观。新觉醒的对象设计师会

试图通过使用了多少模式来衡量设计的质量，虽然 Gamma 等人已经直接警告过这种行为了。对单个对象的重载模式角色导致晶体越来越大，最终代码库成为一个巨型、僵化的石头。

极限编程实践

程序员的宗教战争是传奇的：emacs 和 vi，tabs 和 spaces，VMS 和 UNIX，Java 和.NET。在这个列表中，我们现在应该添加上敏捷方法和传统方法。

不幸的是，任何宗教战争都让热情的少数人走上了极端，导致反对者和对争论模棱两可的大多数人拒绝整个方法。很明显极限编程（XP）就是如此。很多开发人员和开发经理拒绝 XP，认为它散漫或不切实际。（他们恰恰错了，但是这里不解释了。）这是一个耻辱，因为他们摈弃了两条编码方法：重构和单元测试，这两条方法是自从面向对象方法发明以来最有价值的编码方法。

在《重构》[FBB+99] 一书中，Martin Fowler 收集和记录了很多实践方法。他呼吁“改善现有代码的设计而不改变功能”。这绝不是无用功，重构是适应性的关键。它像一股恒定的压力，在保持软件功能的前提下将设计复杂度控制在最小。它打击结晶的传播，并在软件核心类中促进通用水平的提高。

单元测试是重构的孪生兄弟。确实，很多人认为，没有单元测试，就没有重构。没有单元测试的安全保障，改变设计会增加代码库中的随机压力，而且更加可能产生新的 bug 而不是设计的改进。

单元测试的极端形式是 TDD（测试驱动开发）。使用 TDD，你会先写单元测试，然后，将它作为功能规范。你需要编写足够的代码来让测试通过。（亦参见 YAGNI—“You Ain't Gonna Need It!”）一旦测试通过，你就允许重构代码来改进设计，并确保单元测试总是能通过。TDD、重构和 YAGNI 的组合，非常自然地带来高适应性的代码库。

其次，更微妙的影响是通过一致的单元测试产生的。你不应该称一个对象是“可重用的”，除非它被重用。当一个对象经历了单元测试，它就立即用在两个上下文中：产品代码和单元测试本身。这迫使测试下的对象更加可重用。测试对象意味着你需要 stubs 或 mocks 来替代真实对象。这意味着对象必须像暴露属性一样也暴露依赖，从而让它们在产品代码中便于依赖注入。当一个对象在外部环境中需要大量的配置（就像前面提到的 `Customer` 对象），就很难做单元测试了。一个常见并且不幸的反应是停止单元测试这样的对象。更好的办法是减少需要的外部上下文数量。还是以 `Customer` 这个领域对象为例，抽取出它持久性的职责可以减少你提供的外部上下文数量。这让单元测试更容易，也能减少 `Customer` 的结晶大小，从而使 `Customer` 本身更具可塑性。很多这样微小变化的累积效应是深远的。

敏捷数据库

“敏捷”和“数据库”放在一起并不矛盾，而且敏捷数据库的出现并不是偶然的。如果存在一个 IT 领域，比数据库架构定义还更抗拒变化，那它只能是大型机上的 CICS 事务，要知道，这是因为以前的程序员在 20 世纪 70 年代末都去世了，因此无法改变了。

如果行为是第一性的，那么很明显，应用程序的功能变化要适应于数据库架构的变化。为什么数据库通常抗拒变化？首先，并非每个人都同意行为是第一性的！很多公司明示或暗示数据比程序功能更重要。[①]有的公司把数据库架构师封闭起来，远离程序开发人员。每个组都认为自己要胜过其他组。应用程序开发人员希望架构随着程序需求而变；数据库架构师认为他们是数据的保护者——不仅是数据表示而且也是其内在含义。更重要的是，数据库本身常常有多个用户，每个都具有架构和解释的知识。

这是个错误的二分法。行为和数据都不能占据绝对的优势地位。程序员能够而且会创造性地绕过障碍，包括僵化的架构。他们会重载列定义，添加类型标识，或者将数据打包成 XML 字符串放入 CLOB 中。他们会在代码中实现关系，这些关系没有架构的参照完整性检查；他们也会发明基于参照表的编码方案，这些表只在代码中以“枚举”类型存在。用不变的架构改变行为会导致越来越费解的架构滥用，最终导致数据架构师在恐惧中反冲。这种语义污染也传播给了数据的其他用户，使他们使用数据更复杂，也更容易出错。

所以，我认为数据库架构要能改变。现在，你需要考虑如何使这些变化尽可能不带来麻烦。首先关心的是应用程序代码对架构的自身依赖。对象/关系映射让程序的更新变得容易。然而，还有一些限制，特别是关于版本和版本控制。用不匹配 ORM 元数据的数据库架构启动应用程序会产生不可预料的后果。例如，Hibernate 能配置在启动时验证映射文件和数据库结构，但是这很耗时，而且要依赖于你数据库厂商的元数据支持级别。如果启动时不验证，那么架构–元数据不匹配可能带来奇怪的运行时错误，那要么让事务回滚（并报告错误给用户），要么更糟的是导致数据损坏。

每个架构都应该包括一张表，它指示当前的结构版本。这张表可能简单到仅一行一列，只是一个版本号。至少，应用程序应该在启动时检查相应的版本号。根据快速失效原则，在它不能使用数据库结构时应用程序应该拒绝启动。版本号也能用来触发自动架构更新，比如 Ruby on Rails 中的 Migration。[②]版本号在零停机部署时也很重要。因为数据的架构可能改变，即便不是架构本身的直接改变，请也确保版本号的变化在解释中突出。

① 在这些组织能看到数据流图或巨型的 Zachman 框架图。

② 参见 RDoc 中的 `ActiveRecord::Migration`。

18.3 适应性的企业架构

一些架构师想创造一个现代巨人：比如 Forbin（或者 Tron，如果你是我这一代人）。[1]它们工作在无缝的企业机制下，像一个统一的机器一样工作。每个部分都是整体的一个必要零件，非常适合它的角色。程序和程序员在机器内部，服务其需求。我对这样的乌托邦深表怀疑。

这种自上而下的体系架构，通常开始于一个巨型框架，比如 Zachman 或 TOGAF。他们将这种架构看作一个单纯的整体，用系统填充企业架构矩阵的单元。如果这个组有权利，他们会将项目搁置，直到定义了企业架构之后。如果他们没有，他们会在发布投产后付出应有的代价。

这种观点建立在两个有缺陷的假设之上。首先，它假设架构可以被完成。认为企业架构已经“就位”了，也就意味着改变结束了。如果企业架构停止变化，那么公司也就冻结了。

唯一的静止就是死亡。其次，自上而下方法假设公司能经受住时间考验，在企业架构已经定义好的情况下拒绝改变。这是昂贵的实际成本和机会成本。

某种程度上说，我夸张了点。我的本意不是创建一个稻草人，而是强调这种自上而下架构的极权主义倾向。企业，作为一个简单集成、定义良好的整体暗示了其系统的机械观。然而，大型复杂的机械却表现出许多不良的故障模式。它们经常坏掉。当一个环节中断时，它们可能会被削弱。设计和构建这样的架构需要命令和控制层次，这种层次已经一次次地失败了。

最重要的是，这些都需要变化同时跨多个不同的群体发生，这又是一个应用程序的 Conway 法则。由于这些版本锁定系统遍布，企业遭受了严重的问题。当任何对应的部分发生改变时，系统也要相应改变，这都是因为紧耦合的原因。在极端情况下，比如 ESB（企业服务总线）的协议变化，每个参与的系统必须同时升级。试想一下同时部署每个关键任务的企业系统所固有的风险吧！实际上，这些变化，比如大规模协议变化，所带来的成本与风险绝不会发生。ESB 要么僵化，要么以相同的方式被颠覆，比如小规模的静态数据库架构。一旦它足够地过时，ESB 会用新的技术取代它，并声称能克服复杂性的成本。

对企业的机械性比喻困扰了我。机械系统表现了我们不希望组织所享有的这些属性，它们僵化而脆弱。为了避免这些属性，我们从生物和生态的隐喻中获得灵感。我将一个组织看作一个生态系统。人和系统占据了这个生态系统的小格。它们与环境交换资源，主要表现为信息流。生态系统中的单个小格可能有不止一个居民。例如，我曾见过这样的公司，它使用不少于 7 个独立、部分相互操作的 SAP 公司的 ERP 系统。每个系统都有自己的体系，根据公司的收购历史。每个都有自己的组成、食物来源（喂养它数据的系统），以及共生系统（用来输出的系统）。通常情况下，这非常低效，但是健壮。每个实现能够独立于其他的而改变，如果它们是由抱有“一个且唯一”观点的企业架构师开发的话，则个体适应速度更快。

① 如果不知道这些，可以查看http://www.imdb.com。

真正的企业总是比企业架构承认的更糟糕。新的技术从未完全取代旧技术。你会发现一个混杂的集成技术，从简单的文件传输到批处理发布/订阅消息。任何试图创造单一文化的战略都注定会遭受惨痛的失败，这不管是一个集成技术还是一门编程语言。想象一下，一个成功实行单一语言的公司，整个公司由 Ada、Smalltalk、Pascal、C 或其他旧的语言所构建。（一种语言在成为候选之前至少有 10 年了。作为这种模式的范例，我提到 Java，它在许多大型 IT 厂商中被认为是“企业标准”语言。）同样，这种人为的刚性确保了 IT 不会服务于组织的需求。

要问：“这个架构能让 IT 部门更好地响应用户需求吗？”

其实，评价架构最有用的标准是：它能让 IT 部门更好地响应用户需求吗？大多数企业架构并不以此为目标。相反，他们以 IT 部门的需求为目标。（IT 部门好，则公司好。）有可能让企业架构摆脱个别系统之间相互作用的模式。我并不建议彻底的无政府状态，那也不利于让 IT 更好地响应用户需求。相反，有各种混合的压力：预算压力、期限压力、需要的功能、这些功能的直接和间接成本、技术的掌握，和摩擦的消耗。成功的企业架构需要动态地处理这些力量，促进健康组织的交互模式。

系统中的依赖

系统应该表现为松散的集群。在一个松散的集群里，个体的失误对整体的影响好比森林中的一棵树的影响一样。例如，十多个 Apache 实例，少了一个对整体服务并没什么影响。[①]同样，损失一个应用服务器实例不该影响整个应用程序或服务的健康状况。

这意味着，每个服务器角色都一样，或者至少不同的角色能有多个服务。WebLogic 和 WebSphere 都违反了这一原则。它们都需要个别的节点来管理集群。这些节点出现故障会对整个服务带来损坏。

松散集群的成员可以独立地成长或降级。对于集群中成员的激活，应该没有时序上的要求。

一个集群或层的成员应当没有特别的依赖，或对另一层个别成员的认知。依赖应当是虚拟 IP 地址或服务名。直接的成员与成员的依赖就带来硬联系，这阻碍了端点的独立改变。图 18-1 说明了这一概念。调用的应用程序实例仅引用了集群 2 提供的服务名，提供此服务的确切的主机和应用程序不应该让集群 1 的成员所知。

① 假设有足够的容量（参见第 8 章“容量概述”）。

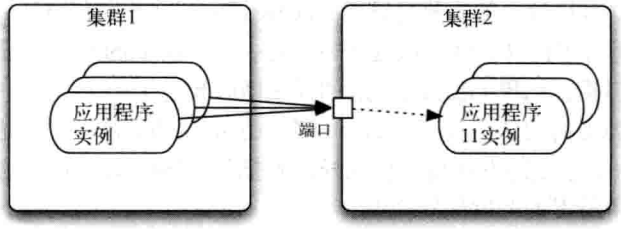


图 18-1 服务的依赖，不是个别实例

集群中的成员不该需要知道集群其他成员的身份。这设置了一个 $O(n^2)$级别的变化需求，使它难于在集群中添加和删除成员。它还鼓励一对多的通信模式，这是容量的杀手。广播通知，比如缓存失效消息，应该通过发布/订阅主题或命令队列来完成。

系统间的依赖：协议

正如你所看到集群对象间的耦合一样，企业系统间的耦合可能引起相同的钙化。系统的紧耦合阻碍了接口两方的变化。而且，良好的架构会拥抱变化，将变化作为驱动改善的引擎，而不是加以控制的野兽。

任何接口都由协议来定义。它可能是底层的，使用套接字；也可能是上层的，比如 CORBA、DCOM，或 RMI；或者在两者之间，比如通过 HTTPS 的 XML。不管哪种协议，接口的两端都必须使用和理解相同的语言。迟早语言是要改变的。那时，可能会有两种情况。如图 18-2，两个系统同时变化。这需要两个系统都停机，因为部署不是瞬间完成的。如果每个系统都只和另一个集成，这可能还可以接受（虽然零停机已经不可能了）。一旦这些系统与其他系统形成网状的集成，那么变化就开始变得站不住脚了。只是支持部署所需的停机时间变得站不住脚了。

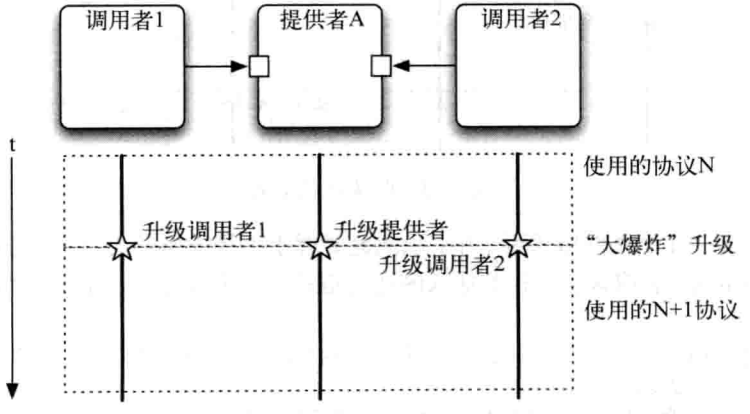


图 18-2 在两端同时变化

更糟的是，每次修订协议，都需要足够的 lead time 来经过开发和测试周期。假设图 18-2 中，系统 Alpha 有一个快速开发团队，使用敏捷开发方法，两个星期的迭代开发。他们将能够在两星期内适应新的协议。但是，如果 Gamma 背后的开发团队超过了两个星期，要么因为他们有更长的迭代时间，或者因为当前的迭代周期已满，那么 Alpha 就必须慢下来以适应 Gamma 的步伐，而 Gamma 可以使用一个季度的发布周期！同样，外推到 Alpha 集成的其他系统，两个星期快速的开发周期会被企业中最慢的开发周期拖慢。[①]

显然，避免这种耦合是有价值的。它对于上市的时间、部署的成本，以及系统的可用性都有不利影响。我们必须设计协议以便接口的两方都能独立变化。解决的方法是协议版本。图 18-3 表明了 Alpha 和 Gamma 如何在不同时间发生改变。显然，存在这样一段时间，那时 Alpha 愿意使用和理解版本 1 或 2，但是 Gamma 只能使用理解版本 1。两个系统为了能协同工作不得不对一个兼容的协议版本达成一致。测试装置（参见 5.7 节“测试装置”）的良好使用这时很有用，它能帮助确保一个系统能够处理它所支持的所有协议组合。协议版本的定义根据实现技术而变。对于低级别的套接字协议，它发送一串字节，这时，版本应该是早期握手的一部分。请注意，协议版本发送方式接下来不可能变化，因此让这部分足够简单以致它不需要改变。对于远程对象协议，系统可能同时暴露多个接口定义，绑定不同的名字。基于 XML 的协议有简单的方式来收发协议版本，并且有更具挑战的内部实现，特别是使用了 XML 到对象的映射工具。[②]

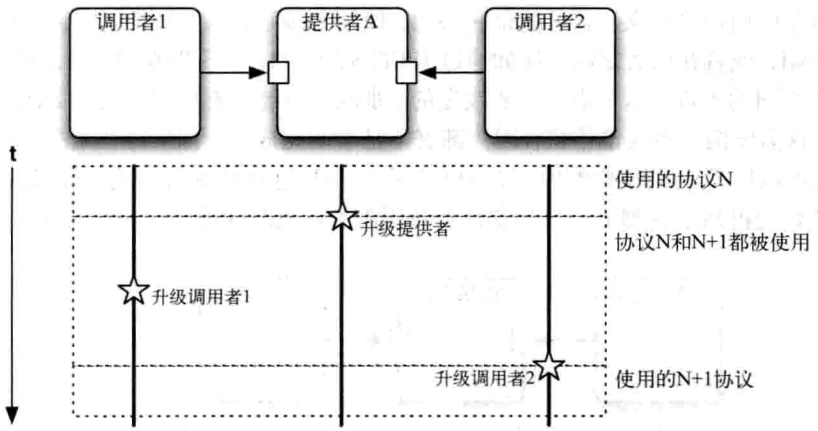


图 18-3 两端独立变化

相同的问题也存在于文件格式版本中，虽然此时握手不是选择方式之一。系统应该准备接受和产生需要的旧格式，它应该自动检测输入的适当格式。当新的格式体现不同的语义时，就可能

① 在 *The Goal* [Gol04]中，Eli Goldratt 描述了这种现象。当时 Alex 看到一支童子军队伍。无论如何施令，最快的人总会被最慢的人给拖慢下来。

② 例如，JAXB 根据对象产生 XML 架构，反之亦然。维持两套相同名字的类，但是绑定不同的结构确实困难。这是我不赞成这些工具的理由之一。

有多个映射问题。定义格式的变化需要定义如何解释旧版本格式。

图 18-3 中，如果只是发生在系统之一的部署期间，重叠的时间可以按照分钟或小时的顺序。另一方面，如果 Gamma 的开发人员的发布周期是一个季度（或更糟），那么它可能是一个漫长的过程。

系统间的依赖：数据库

集成数据库——不要这样做！我是认真的！即便没有视图。即便没有存储过程。采用它的级别，然后围绕数据库包装一个网络服务。然后让网络服务冗余，并且通过虚拟 IP 访问。建立一个测试工具来验证，网络服务停掉时会发生什么。这是一个企业集成技术。把手伸到了另一个系统数据库是太过了。

没有什么比让其他系统来戳破它的胆更摧毁系统的适应能力了。数据库“集成”纯粹是罪恶。它们暴露了系统内部运行的重要细节，违反了封装和信息隐藏原则。它们鼓励结构和语义上的不适当耦合。

更糟糕的是，挂起数据库的系统不能信任其数据。数据行可以被其他实体添加或修改，而拥有者在内存中有映射这些行的对象。重要的应用逻辑可能被绕过，导致非法或无法到达的状态。[①]

多个系统访问的数据库成为刚性的连接点。只要当架构改变时，每个系统必须同时升级。（并且，最可能的结果是架构永远不改变。）

不过，有时候其他系统确实需要访问大量他人的数据。或许它用于报告或商业智能工具，或许用于分发给渠道合作伙伴或财务分析。在这些情况下，我们能找到一些替代品，它们不会招致戳另一个系统胆的罪恶。

交易大厅之外，大多数任务的目的并不要求对生产数据立即做第二次的访问。如果需求表明这种及时性，请在分析和决策过程中检查内在的延迟。（参见 17.8 节“支持流程”。）

奇怪的是在行动之前这些数据可能陈旧达数小时之久。这时，获得生产数据有多重要呢？前一个小时的快照就会是及时的。（如果这个反馈过程只需要几秒钟的时间，那么情况就完全不同了。）

从产品传输大量数据有时候是必要的。这时，使用提取–转换–加载（ETL）工具来处理物理和语义映射。一旦转换被定义，它们就能被调整，并根据配额来避免产品容量的风险。运行提取的 ID 能被设置允许“选择”，但是不能“插入”、“删除”或“更新”。Oracle 的 materialized views 在传输大量数据库时也服务良好。根据存储的架构，你也可能使用所谓的业务连续性卷（BCV）来做游戏，将数据从一个 SAN 移到另一个。

① 真正的数据库偏执狂会认为数据库本身应该是业务逻辑的资源库。我以前曾听说过。原则上，这似乎也合情合理。数据库有图灵完备语言可用。实际中，它是在错误的地方的错误逻辑，使用了错误的工具和变化流程。

18.4 发布应无害

我最喜欢的一个零售系统，它的发布流程能够匹敌 NASA 的启动流程。它从下午启动，运行到凌晨。在早期，要 20 多个人积极地参与发布过程；现在，这个过程已减少到不到 12 个人。你可以想象，任何需要很多人参与的流程都需要详细的规划和协调。因为每次发布都是任务艰巨，所以他们一年中这样做得并不多。由于就这么几次发布，每次都显得比较独特。这种独特性需要对每次发布都进行特别的规划，这使得发布更加痛苦，进一步阻碍了发布的频率。

发布应该是和剪头发（或者说编译内核，如果你是个灰帽 UNIX 黑客的话）一样大的事情。关于敏捷方法、精益开发和增量投资的大量文献，都认为经常发布能让用户喜悦并产生商业价值。[①]然而，说到产品运营，频繁发布有个额外的好处，它能让你在发布和部署上做得更好。

在第 17 章我们讨论说一个封闭的反馈循环对于系统改进是必不可少的。反馈循环操作得越快，改进也越准确。这需要经常地发布产品。新功能的频繁发布也让公司超过其竞争对手，在市场中树立自己的地位。

通常，发布成本很高，并且会引入风险。我前面描述的这种人力和协调工作很少能维持一年发布三四次，绝不可能达到一年二十次。一种容易实现但有害的解决方案是减慢发布。就像因为害怕牙疼就很少看牙医一样，这种解决方案会让问题加剧。正确的方式是减少发布的代价，从流程中除去人的干预，使整个过程自动化并且标准化。

部署成本太高

除了产品公司，发布软件的成本几乎从不考虑在预算中。规划、分析、开发和测试都名正言顺地获得了认可，但是通常没有意识到发布本身常常有大量直接和间接成本。

直接成本主要来自发布执行期间的劳动力。除了测试，还在以下几个方面有任务。

- 配置管理

创建发布分支，标记代码库，创建发布版本。

- 文档

发布说明、新建的或更新的培训材料

- 营销沟通

更新营销材料和公司内外的公告

① 如果《敏捷软件开发》[Coc01]、《精益软件开发》[PP03]、*Software by Numbers* [DCH03]、《极限编程解析》[Bec00]这些书还不能说服你，那就没办法了。

- 部署

计划、执行和验证部署

- 支持

紧急 bug 修复，更新到监测系统，更新到手册

发布的最大间接成本来自发布期间的停机。只在“营业时间”运营的系统很罕见。或者，换个角度看，“营业时间”已经覆盖到一天 24 小时了，那些每晚安排“系统维护时间”的日子已经一去不复返了。

来自运营的观点

我最喜欢的运营经理之一经常提醒人们要"假设意图是正面的"，也就是说，当个人或者团队之间发生冲突，要假设每一方之所以这样都是出于为了公司好的愿望。在最坏的“我们”对“他们”的冲突之中，可能看起来像“就是要和我们过不去”的情况，但是这很少发生。冲突的发生通常是因为，不同的小组按照不同命令运营。

在说到发布的频率时，假设意图是正面的很有必要。在客户和开发团队看来，新版本能产生积极的改变。开发组看见新版本时，看到的是需求被满足了，bug 被修好了，设计也得到提升了。客户看见新版本时，他们看到的是能创收的功能，新版本能留住用户，以及可以比得上或者超过他们的竞争对手。对客户和开发团队来说，改变带来提升。

当运营部门看到一个新版本时，他们看到的是潜在的会失败的新模式、需要换新流程、新的监督过程。他们发现旧的监督过程已经不再适用，也意识到新的版本可能需要更多的“核心四资源”：CPU、内存、存储器、带宽。除了以上这些，他们还看到了风险——部署本身有风险、新版本有风险，而且由于和现有的应用程序代码库比起来，新版本处于未知状态，这也存在风险。

新的版本对于他们意味着这些，所以客户、开发团队和运营经理都是对的。

运营很可能根据计划外的停机时间来计算可用性，因此 99.5%的可用性只是意味着系统每个月有少于 216 分钟的意外停机。这可以说是个借口。系统可能在改变窗口期间停机 5 个小时了。但是系统的用户会关心停机是计划内的还是计划外的吗？不会！对于用户来说，停机就是停机。如果我不能完成工作，系统就是停机了！永远记住用户通常不会了解开发或运营的工作日历，他们只关心自己的日历。

定时发布

为什么我们如此强调发布日期？对于微软或 Adobe 这样的产品供应商，发布日期事关重大。它们提前一年宣布，伴随着巨型产品启动事件、媒体的关注、客户和销售渠道的大规模计划和准备。没人会在午夜的时候排队，从其他的 99.9%的人手中得到最新的软件版本。

大多数时候，我们的客户甚至不知道预定的发布日期。对于基于 Web 的软件，发布改变了一天，对于客户来说无关紧要。如果发布晚了一天或一个星期，客户不会知道也不会关心。客户关心的是系统可用，并且没有太多的错误。

> 不要用任意的发布日期去危及客户。

然而，我曾见过公司在没有准备就绪之前就急于发布产品。曾经有一次，我在发布当天的下午 4 点参加决定是否发布的会议。QA 告诉我发布版本还没通过测试，但是再给一个小时他们会知道更多。在这样的情况下，委员会竟然同意了发布！我反对，说发布还没就绪，除非它通过了 QA。为什么要用几个月前随意选定的一个日期作为发布日期，并且还要危及客户呢？答案很明显，他们根本不关心什么对客户最好！

零停机时间部署

在高可用的服务器和高可用的架构下，我们就不再能接受硬件维护的停机时间了，那么，为什么在软件改变时我们还要忍受它呢？

不管怎样，几乎每次发布时都需要停机。不管是一小时还是一天，称它为“计划内停机”并不会省掉成本。如果公司的停机成本是每小时 10 000 美元，那么 4 小时的部署成本就是 40 000 美元，不管是不是在运营日历内。

为什么应用程序发布需要停机呢？讽刺的是，这是为了增加运行时间由同样的架构功能导致的：冗余。事实上，多台服务器处理请求，意味着在部署期间，有些服务器上的版本是新的，有些是旧的。在这个重叠时间，版本 *N* 和版本 *N*+1 都存在。如果软件完全是自成一体的，那么两个版本同时存在并不会有问题。当然，企业应用程序很少是自成一体的，它们要使用数据库、网络服务和搜索引擎。对于网站来说，它们产生 URL，会引用样式表、JavaScript 文件和多媒体文件。所有这些外部引用都提高了版本冲突的机会。最简单的方法是在部署的时候停机。

然而，有可能在较长的时间内执行部署，几天甚至几个星期，这可以避免版本冲突，并让两个版本同时存在。让一个零停机部署结构化需要开发与运营的合作，这是它很少能实现的原因。然而，稍微使用一点前端架构，你就能消除部署的停机时间。

这个关键就是按阶段地分解部署。并不是立刻添加、改变和删除一些东西，比如数据库的列

和表、约束、服务，而是尽早添加新项目，并确保旧版本代码的前向兼容性。然后，发布后回滚，移除不再引用的东西，然后添加会打破旧版本的新约束。下面几节会详细描述这些阶段。

扩展

第一步是添加新的“东西”，它包括基于 URL 的资产、网络服务端点、数据库的表和列，等等。所有这些都能添加，并且在一定条件下不会让旧版本的软件崩溃。

基于 URL 的资源，比如样式表、图像、动画或 JavaScript 文件，应对每个新版本给予一个新的 URL。如果应用程序产生了引用`/static/styles.css` 的网页，那么对 `styles.css` 做任何改变都可能带来版本冲突。相反，如果应用程序引用的是`/static/1.1/styles.css`，那么部署`/static/1.2/styles.css` 时就没有冲突了。

对于网络服务，接口的每次修改都应该赋予新的端点名字。同样，对于远程对象接口，对每个版本定义新的接口名（例如，在接口名后加一个数字）能确保旧版本获得想要的接口，同时在新版本中也获得想要的接口。一个简单的实现类能容纳多个接口定义，只要所做的变化仅添加或删除了暴露的方法。更改现有方法的语义需要不同的类定义，通常是通过一个适配器或一个释放出的超类。

关于基于套接字的协议，协议本身应该包含一个版本标识符。这无疑要求接受方必须在发送方之前更新，也暗示了接收方必须支持协议的多个版本。如果有其他接受方的调用者，那么这有助于与这些系统降低耦合。为收到这些调用的服务器部署新版本的代码，可以作为它自己的“递归零停机部署”。

如果不切实际地支持多个协议版本，另一个选项是在负载平衡器的不同端口上定义多个服务池。那么调用程序的旧版本将会获取映射与接收程序的原始版本端口号，同时调用程序的新版本会获取映射于接收程序的新版本端口号。

到目前为止，大多数的冲突和最棘手的冲突都会在数据库中出现。架构的更改很少向前兼容，它们也从不是偶然发生。不过，有可能将架构改变分成几个阶段。在“扩张”阶段，先添加表和列。任何 NOT NULL 的列被添加为可为空，因为旧版本不知道如何填写这些。然后，在清理阶段，再添加约束。这也适用于参照完整性法则。它们不能在扩展时添加，因为旧版本会立即违反关系。

因此，即便旧版本不因为新的数据表和数据列而出故障，它也不会做任何有用的事情。新版本很可能期望这些有意义的数据。它甚至可能包含集成脚本来用旧表中的数据填充新的表和列。应用程序很可能只在数据库中加入新的行。[1]你能使用这个技巧来传递新表和列的数据：触发器。

① 如果应用程序使用存储过程添加条目到数据库，然后存储过程应该被更新，以便在旧版本应用程序调用时填充数据。

假设新的结构能从现有数据来填充，并且对如何使用正向集成脚本有示例，那么触发器就能做相同的事情，一次集成一行数据。因此，旧版本会创建新版本使用的数据。毕竟，新版本也会创建数据，它会使用新的结构。所以，如果不采取措施，旧版本会认为新版本的数据损坏或不完整。触发器也能弥补这个问题，它能根据新数据来填补旧结构。

当然，为了对此做准备，任何 SQL 中的插入或更新语句必须显性地关于列和值。SELECT * 也不太可能有所帮助。再次，ORM 工具能起作用。它们机械地产生 SQL 语句，这些语句包括具体列的查询、插入或更新。任何辅助查询，比如商业智能或报表，也应该具体到感兴趣的列。如果你从一开始就这样做，就不难做到了。尝试替换每个 SELECT *来做发布并不太管用。

推广

有了“扩展”阶段的准备，新软件在服务器上的推广应该很容易了。这可能要花几小时到几天时间，取决于你多谨慎地完成它。例如，这可能包括让一堆服务器在代码库上“烘培”一天或更久。（当两个版本都使用时，在负载均衡器上创建两个服务池能有助于一组服务器的请求或会话失效备援，这组服务器和原始会话使用相同的版本。）

如果部署需要停机，那么就有压力，以图在最短时间完成这一阶段。排除了压力，就有足够的时间来有序关闭。这应该遵循良好的做法来干净地关闭和启动，从而避免用户在突然关闭时的无奈。[①]（参见 14.3 节“启动和关闭”。）

清理

当新版本烘培了足够长时间，能被接受后，就是时间清理了。这包括删除桥接触发器和额外的服务池。任何不再使用的列或表都可以被删除。旧的静态文件也能删除。

这时，所有的应用服务器都运行新版本。是时间将需要的列转变为 NOT NULL 了，同时添加引用完整性约束（数据库中执行的约束可能对 ORM 层带来大问题）。此时也能去掉不需要的列和表了。

18.5 总结

变化是软件的决定性特征。这种变化，即适应性，从发布时就已经开始。发布是软件真正生命的开始，发布之前的一切是妊娠。系统要么随着时间成长，适应变化的环境，要么衰退至死，因为它们的成本超过了其益处。

① 主要是丢失的会话状态。我从未见过完美的失效备援。

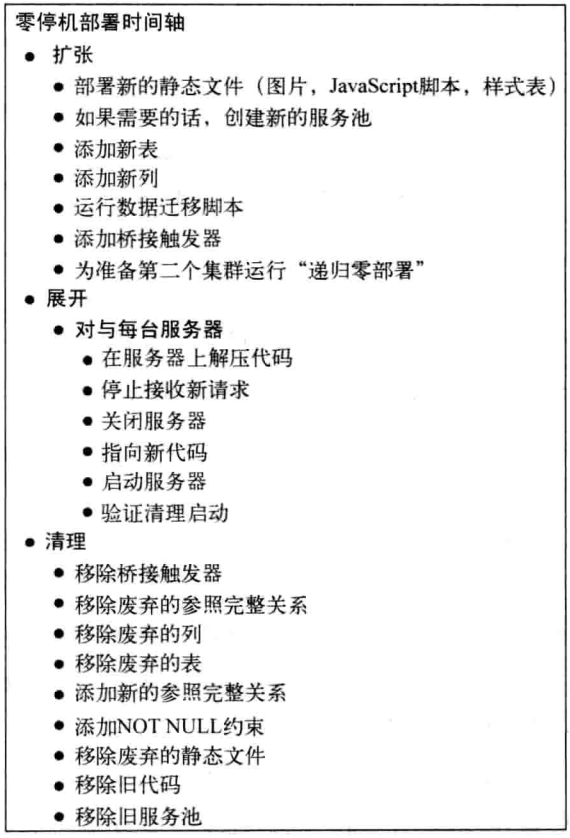


图 18-4 零停机部署：详细时间表

参考书目

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 2000.

[BF01] Kent Beck and Martin Fowler. *Planning Extreme Programming*. Addison-Wesley, Reading, MA, 2001.

[Chi01] James R. Chiles. *Inviting Disaster: Lessons From the Edge of Technology*. Harper Business, New York, NY, 2001.

[Cla04] Mike Clark. *Pragmatic Project Automation. How to Build, Deploy, and Monitor Java Applications*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2004.

[Coc01] Alistair Cockburn. *Agile Software Development*. Addison Wesley Longman, Reading, MA, 2001.

[DCH03] Mark Denne and Jane Cleland-Huang. *Software by Numbers: Low-Risk, High-Return Development*. Prentice Hall, Englewood Cliffs, NJ, 2003.

[DeM95] Tom DeMarco. *Why Does Software Cost So Much?* Dorset House, New York, NY, 1995.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Longman, Reading, MA, 1999.

[Fow96] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison Wesley Longman, Reading, MA, 1996.

[Fow03] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley Longman, Reading, MA, 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object- Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[Gol04] Eliyahu Goldratt. *The Goal, 3rd ed*. North River Press, Great Barrington, MA, 2004.

[JG06] Dion Almaer Justin Gehtland, Ben Galbraith. *Pragmatic Ajax: A Web 2.0 Primer*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2006.

[Koz05] Charles Kozierok. *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference*. No Starch Press, San Francisco, CA, 2005.

[LC99] Domenico Lepore and Oded Cohen. *Deming and Goldratt: The Theory of Constraints and the System of Profound Knowledge*. North River Press, Great Barrington, MA, 1999.

[Lea00] Doug Lea. *Concurrent Programming in Java, Second Edition: Design Principles and Patterns*. Addison-Wesley, Reading, MA, 2000.

[LW93] Barbara Liskov and Jeannette Wing. Family values: A behavioral notion of subtyping. Technical Report MIT/LCS/TR-562b, 1993.

[Moo91] Geoffrey A. Moore. *Crossing the Chasm*. Harper Business, New York, NY, 1991.

[MP06] Mary and Tom Poppendieck. *Implementing Lean Software Development: From Concept to Cash*. Addison-Wesley, Reading, MA, 2006.

[Nor88] Donald A. Norman. *The Design of Everyday Things*. Doubleday/Currency, New York, NY, 1988.

[Pet92] Henry Petroski. *The Evolution of Useful Things*. Alfred A. Knopf, Inc, New York, NY, 1992.

[PP03] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit for Software Development Managers*. Addison-Wesley, Reading, MA, 2003.

[Ric04] Chet Richards. *Certain To Win*. Xlibris Corporation, Philadelphia, PA, 2004.

[Sen90] Peter Senge. *The Fifth Discipline: The Art and Practice of the Learning Organization*. Currency/Doubleday, New York, NY, 1990.

[She97] Michael Shermer. *Why People Believe Weird Things*. W.H. Freeman and Company, New York, NY, 1997.

[Ste93] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Reading, MA, 1993.

[VCK96] John M. Vlissides, James O. Coplien, and Norman L. Kerth. *Pattern Languages of Program Design 2*. Addison-Wesley, Reading, MA, 1996.

[WID+98] Craig Wisneski, Hiroshi Ishii, Andrew Dahley, Matt Gorbet, Scott Brave, Brygg Ullmer, and Paul Yarin. Ambient displays: Turning architectural space into an interface between people and digital information. *Lecture Notes in Computer Science*, 1370:22, 1998.

http://citeseer.ist.psu.edu/wisneski98ambient.html.